

Introduction

This document describes the inner workings of IBM OS/2 v 2.0 kernel, based on the joint research taken by IBM and Microsoft during the design stage for OS/2 v 2.0. It is provided for informational purposes only and is not meant to be the complete and authoritative source for the whole kernel design specification.

The information in this document applies to the following products:

- IBM Operating System/2, Standard Edition 2.0
- Service Pack XR06055
- IBM Operating System/2, Version 2.01

For further reading, consult the following IBM documentation:

1. **Cruiser Performance Specification, Rev. 2.3**, IBM Internal Memo - ITSC Boca Raton, Jan 24 1991.
2. **OS/2 v 2.0 Perfview OEMI Document**, IBM OS/2 v 2.0 OEMI distribution, Oct 31 1991.
3. **OS/2 v 2.0 Technical Compendium**, IBM Red Book GG24-3730 (1992).
4. **New APIs and functionality in XR06055**, IBM Technical Notes - Boca Raton, Oct 3 1992.
5. **Installable File Systems for OS/2 version 2.0**, IBM OS/2 v 2.1 OEMI distribution.
6. **OS/2 v 2.1 Performance Improvements**, IBM Internal Memo - Boca Raton, May 10 1993.
7. **OS/2 v 2.1 System Trace Facility Event Descriptions**, IBM OS/2 v 2.1 OEMI distribution.
8. **Merlin presentations**, IBM Support Education for Merlin - Austin, September 1996.
9. **The OS/2 Debugging Handbook**, IBM Red Book SG24-4640 (1997).

Note: This document was originally created for the publishing and code documenting system based on SCRIPT/VS. The conversion to IPF required to drop certain content, including structure and entry point listings. Consult the relevant header files for further information. Sorry for this inconvenience.

In this document, certain versions of OS/2 may be referred to by their code names. The following naming convention exists:

- Any version of OS/2 is sometimes referred to as "CP/DOS".
- Presentation Manager v 1.1 is referred to as "Winthorn".
- OS/2 v 1.2 is referred to as "Sloop".
- OS/2 v 2.0 is referred to as "Cruiser".
- A provisional WPS-enhanced Cruiser is referred to as "Yawl".
- OS/2 v 2.01 is referred to as "Riker I".
- OS/2 v 2.1 is referred to as "Borg".

This edition was compiled by Andrew Belov on December 30, 2003. It is still work-in-progress, therefore several sections may be misaligned and some automatically-extracted text may be hardly readable. Comments are welcome at andrew_belov@newmail.ru.

What's New

This chapter records the key changes on the Sloop-Cruiser track.

Amendment, Nov 6 1987

NE-EXE format for Sloop.

Update, Oct 10 1988

Tasking Architecture.

Update, Nov 2 1988

Page Manager.

Amendment, Nov 10 1988

Page Manager - KPI.

Update, Nov 15 1988

SFT description.

Update, Dec 1 1988

IFS API for Cruiser.

Amendment, Dec 20 1988

Page Transition chart.

Update, Mar 14 1989

Preliminary notes on VDDs/VDHs in Cruiser.

Amendment, Apr 7 1989
Move infoseg description from tasking to page manager.

Update, May 17 1989
Program Loader interfaces.

Update, Jun 26 1989
User-mode debugging.

Update, Jul 11 1989
Interrupt Manager.

Amendment, Aug 31 1989
Clarifications to the debugging chapter.

Update, Sep 14 1989
Swap Manager.

Update, Sep 15 1989
Multitasking in Cruiser.

Amendment, Sep 29 1989
Show memory allocation for MTE.

Update, Oct 19 1989
Overview of SAVDM/MAVDM.

Amendment, Nov 19 1989
Multitasking.

Amendment, Nov 22 1989
Master index.

Update, Dec 18 1989
NE-EXE for Sloop (final),
LE-EXE for Cruiser (preliminary).

Amendment, Jan 23 1990
Page Manager - aging/performance.

Update, Jan 25 1990
Selector Component - initial release for Cruiser

Amendment, Mar 22 1990
Final touches to the debugging chapter.

Amendment, Apr 5 1990
Reviewed the selector management chapter.

Amendment, Jun 19 1990
VMM/Program Loader.

Amendment, Jul 10 1990
Multitasking (final).

Amendment, Jul 29 1990
VDH services.

Update, Aug 1 1990
Included PerfView hooks.

Amendment, Aug 20 1990
Page Manager (final).

Amendment, Aug 22 1990
DPMI Exception Handling.

Amendment, Aug 24 1990
LE-EXE for Cruiser (final).

Amendment, Sep 18 1990
MVDM component-level overview (final).

Amendment, Sep 19 1990
VMM interfaces (final).

Amendment, Sep 26 1990
V86/VMBOOT overview (final).

Amendment, Oct 10 1990
MVDM (final).

Amendment, Nov 29 1990
PerfView (final).

Update, March 1991
LX-EXE for Cruiser (preliminary), obsoletes LE-EXE.

Amendment, Apr 4 1991
LX DLL termination.

Amendment, Apr 12 1991
VDPMI.

Amendment, Jun 1 1991
Separate Kernel Debugger document for SDK.

Amendment, Aug 8 1991
VDM Interrupt Handling.

Amendment, Jan 22 1992
LX-EXE format for Cruiser (final).

1st Edition, Jan 22 1992
Complete file set for Cruiser

Amendment, Jun 3 1992
LX format amendment for Riker I.

2nd Edition, Oct 10 2001
Preliminary INF distribution.

Amendment, Jul 29 2003
Cleanup in trace sections, corrected some terms.

3rd Edition, Nov 25 2003
Major cleanup for a new INF distribution.

Device Drivers

This page is intentionally left blank.

Interrupt Manager

This page is intentionally left blank.

Interrupt Manager Architecture

This page is intentionally left blank.

Problem Description/Objectives

The Interrupt Manager is responsible for the routing of hardware interrupts to device drivers and maintaining the physical 8259 PIC state. Performance is a major requirement for hardware interrupt routing. In this document "hardware interrupt" and "interrupt" mean the same thing, but "simulated interrupt", "interrupt simulation" or "hardware interrupt simulation" refers to sending a VDM an interrupt.

The Interrupt Manager provides the following:

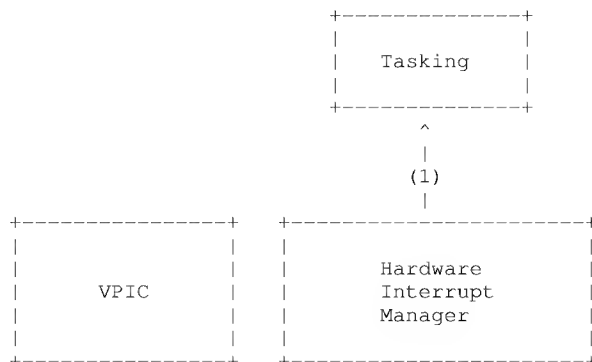
- High performance hardware interrupt routing to device drivers.
- Compatible 286 based OS/2 Interrupt Manager device help interfaces (dh_SetIRQ, dh_UnSetIRQ, dh_EOI, dh_SetROMVector).
- Support for level-triggered/interrupt sharing devices (for PS/2 like machines).
- Support for edge-triggered interrupt devices (for AT like machines).
- Efficient Interrupt Manager packaging/adaptation for third party extensions.

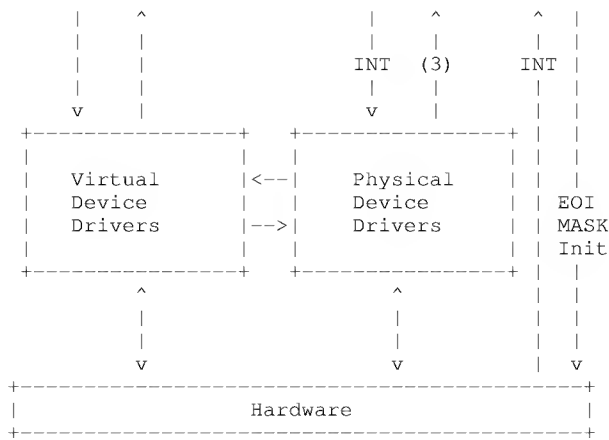
The hardware interrupt router's requirements are the same as the 286 OS/2 based systems, but without the DOS support. It dispatches hardware interrupts to the device drivers registered for this interrupt. The interrupt router does not support any real mode, 3XBOX or MVDM services. The interrupt router is responsible for maintaining the "interrupt time" context for the system. Rescheduling and forced actions are checked on every hardware interrupt by the router to allow preemption of CPU bound tasks.

Rescheduling needs to be done on every hardware interrupt to insure that high priority processes (made runnable in a device driver's interrupt handler) are scheduled as soon as possible before any other user code is executed. Forced actions have to be serviced before more user code is executed; otherwise, user code would be executed after a process was terminated (via the kill forced action) and OS/2 's architecture depends on this not happening. If a disk interrupt made runnable a process with a priority higher than the interrupted task and we rescheduled only on timer interrupts, the high priority process would not be run soon enough to meet OS/2 's dispatch latency requirements.

Solutions/Justification

The interrupt router is not changed much from the 286 based OS/2 interrupt manager. The 3XBOX specific code of the old interrupt manager will be removed or greatly simplified, but the basic protected mode device driver support will remain intact. PDDs always receive the hardware interrupt from the router and can pass the interrupt to a VDD. There must be a PDD to register and receive hardware interrupts.





Hardware Interrupt Management

NOTES:

1.
 - Resched
 - Force Flags
 - InterruptLevel
2.
 - SetIRQ devhlp
 - UnSetIRQ devhlp
 - PhysEOI devhlp
 - SetROMVector devhlp

Interrupt Manager Design

This page is intentionally left blank.

Design Overview

Hardware interrupts are received by the interrupt router through an interrupt gate in the IDT. The interrupt router increments "InterruptLevel" and switches to the interrupt stack if interrupts are not nested. "InterruptLevel" is the mechanism used to indicate the interrupt time state to the rest of the system. The old "current IRQ number" is saved and set to the incoming IRQ number. The "current IRQ number" is used by the PhysToVirt service to choose a protected mode alias selector.

If an device driver has registered an IRQ handler via the SetIRQ devhlp interface, the interrupt router sets up the device driver's DS register and calls it's interrupt handler. If interrupt sharing exists, all the interrupt handlers registered for this IRQ are called until one indicates that it has handled the interrupt.

If there isn't a PDD registered to the incoming IRQ, the IRQ is masked and sent EOI and no device driver is called. Devices that generate interrupts must have a PDD. See the section on unknown device interrupts. PDDs never manipulate the 8259 PIC directly. All PIC operations must be done thru the interfaces the interrupt manager provides.

After the interrupt router is returned to, "InterruptLevel" is decremented and the stack is switched back to the user's stack if this is the last level of nesting. "Current IRQ level" is restored to the previous value. The tasking component's "resched" flag is checked to see if any rescheduling is needed. If the code interrupted is not ring 0 code and interrupts are not nested, we go into kernel mode to do rescheduling, forced or delayed actions. After this is done, we finally return to the interrupted code.

The interrupt router does no VDD/PDD/VDM contention resolution. The physical and/or virtual device detects and resolves device contention. The interrupt router handles no 3XBOX specific activities like changing the 8259 PIC state when the 3XBOX is put in the foreground or background, or checking the 3XBOX interrupt vector table on every interrupt to catch changes by DOS applications.

The major change to the 286 based OS/2 interrupt router design is the removal of any 3XBOX specific functionality. All the 3XBOX interrupt related functions are provided by the VPIC module.

Interrupt Manager Initialization and Packaging

The hardware (8259) dependent sections and interrupt router is packaged in a separate "base-like" physical device driver separate from the VPIC virtual device driver package. This allows easier third party extension for different hardware for both the interrupt router and VPIC. Removes the hardware dependent code from the kernel.

Imported Interfaces

- "Resched" variable

This is the mechanism for the scheduler to inform the interrupt manager that some rescheduling needs to be done on the way out of the interrupt manager on a hardware interrupt. This is also the mechanism used to force dispatching (thru Enter/Exit_Kmode) to post pending signals, etc.

- Enter_Kmode/Exit_Kmode routines

This routines are used to reschedule or service forced actions (like signals, termination events).

- "Indos" variable

This informs the interrupt manager that we are in kernel mode and no rescheduling or forced actions can be serviced on the way out of the interrupt manager on a hardware interrupt.

- IDT entries for the hardware interrupts.

The interrupt manager needs to put it's handlers in the IDT entries for each of the IRQs.

Design Constraints

- Device drivers may not manipulate the 8259 PIC directly.
- Existing installable physical device drivers that manipulate the 8259 PIC directly will not be supported.
- Software interrupt handlers registered via the dh_SetROMvector service will not be called.

Interrupt Manager Implementation

This page is intentionally left blank.

Interrupt Manager Design History

Alternate Hardware Interrupt Routing

1. Have the physical and virtual device tell the interrupt manager who will get the interrupt.
 - + This is pretty fast.
 - May require that the VDD always be present.
 - There is interfaces and protocols to define on how to handle this switching.
2. Chain thru each of the physical or virtual device drivers that have hook the IRQ until one says the interrupt was handled.
 - + Lets the device drivers figure out where the will interrupt go.
 - Requires that the VDD always be present.
 - This is too slow and messy.

Alternate Interrupt Manager Packaging

1. Package just the hardware dependent portions in some kind of loadable module, not the whole interrupt router.
 - + The hardware independent portions interrupt router can now remain in the kernel and only a small hardware dependent module needs adaptation. This would make third party extension for different hardware even easier because the whole interrupt router wouldn't have to be understood and adapted.
 - + The VPIC would not have to be loaded in a protected mode only system.
 - + The VPIC could be loaded after the base device drivers are loaded with the protected mode loader (as in an installable physical device driver).
 - Interrupt router performance may suffer because of the far calls needed to call a separate module.
 - This hardware dependent portion of the interrupt manager router would have to be loaded before any base device drivers.
2. The hardware (8259) dependent sections and interrupt router could be left in the kernel (like it is in 1.0/1.1) and the PIC virtualization and interrupt simulation be packaged in an installable VPIC VDD.
 - + The VPIC would not have to be loaded in a protected mode only system.
 - + The VPIC could be loaded after the base device drivers are loaded with the protected mode loader (as in an installable physical device driver).
 - + Easier third party extension for different hardware for the VPIC.
 - Leaves hardware dependent code in the kernel (harder to adapt the interrupt router for different hardware).
3. The hardware (8259) dependent sections, interrupt router, PIC virtualization and interrupt simulation could be packaged in the installable VPIC VDD.
 - + Easier third party extension for different hardware for both the interrupt router and VPIC.
 - The VPIC would always have to be loaded.
 - The VPIC would have to be loaded before any base physical device drivers.

V86 Mode Kernel Stack

How to handle hardware interrupts during V86 mode kernel code. There is a problem regarding interrupting V86 mode kernel code running on the TCB kernel stack. When hardware interrupt happens while in V86 mode on the TCB kernel stack, the beginning of the TCB kernel stack is corrupted because of the privilege level transition (V86 mode code is considered to be ring 3 code) and the TSS ring 0 stack normally points to the TCB kernel stack. Some solutions to this problem are:

1. Kernel code will never run in V86 mode; always switch to protected mode.
 - + The problem goes away because we are always at ring 0 while on the TCB stack.
 - + No interrupt time performance degradation.
 - A large amount of currently real mode kernel code would have to be converted to run in protected mode.
2. While in kernel code that is running in V86 mode, use a secondary V86 mode stack. The TSS ring 0 stack points to the normal TCB stack. When a hardware interrupt comes in the normal TCB stack is used and the V86 mode stack isn't trashed.
 - + No interrupt router performance degradation.
 - Extra space wasted in the PTDA (for the V86 mode stack).
3. The TSS points to an secondary V86 mode stack when an VDM is running. When a hardware interrupt is received the IRET frame is copied to the interrupt stack and the interrupt stack is switched to.
 - Slows down interrupt router performance considerably.
 - Extra space wasted in the PTDA (for the V86 mode stack).

The second proposal is the preferred solution. The "DOS emulation" layer that runs in V86 mode will have it's own stack.

Special Mask Mode

Do we want "special mask mode" interrupt architecture? "Special mask mode" handles a hardware interrupt as follows: the incoming IRQ is physically masked and EOI is sent to the 8259. This is a big departure from the previous 286 based OS/2 products which leaves the IRQ unmasked and lets the device drivers send EOI (via a devhlp) when finished with it's interrupt handling. PDDs or VDDs will never manipulate the 8259 PIC directly. All PIC operations must be done thru the interfaces the interrupt manager provides.

The advantages/disadvantage of "special mask mode" are:

- + Interrupts can be enabled sooner (after router sends the EOI) and device drivers with long interrupt handlers won't keep other IRQ's from coming in. This provides better interrupt dispatch latency time.
- Existing OS/2 device drivers are not suppose to manipulate the 8259 PIC directly, but the ones that do (the network device driver is the only one known at this time) will not be supported.

Special mask mode does not have enough measureable performance advantage and causes incompatibilities with existing device drivers. It will not be used.

Device Driver SIO

The Device Driver SIO supports direct access reads and writes to a swapping disk partition (type not yet decided). This type of partition will be recognized by OS/2 and no access other than swap operations will be allowed. During initialization, the Device Driver SIO will attempt to open the specified partition for swapping. This consists of querying the actual device driver for the partition type and size, and informing it that the partition will be used for swapping. Once the partition is opened, all of the swapper data structures will be set up, and never need change size again.

Unlike the SFT-based system, conversion from swap id to absolute sector number will not be done in the SIO, but rather in the device driver itself. This is because at the physical level the disk is not simply a continuous array of sectors but instead are addressed by track and sector. Since each 4 kbyte page occupies 8 512 byte sectors, we must guard against the possibility of a page's being assigned to sectors on different tracks. If the page were split in this manner the I/O operation would have to be done in multiple pieces, adding complexity to the device driver. Current disks have 17 sectors per track, but this is not guaranteed. If the SIO were to assign pages to sectors blindly, over half would span track boundaries. To avoid this, the device driver (which knows about sector layout) will be enhanced with knowledge of swap ids and will make the translation between swap id and track:sector disk address. With the 17 sector tracks used on many disks, the translation would be $\text{track} = (\text{swap id} \gg 1)$ and $\text{starting sector} = (\text{swap id} : 1) \ll 3$. This would make 16 out of the 17 sectors on each track (roughly 94% of the total disk space) available for holding data, with the remaining 6% of the disk wasted. Calculations for disks with a different number of sectors

per track can be carried out in a similar manner and can be expected to yield roughly comparable results.

During initialization the device driver will be queried as to whether or not it uses DMA and whether it takes physical or virtual addresses. This information will be passed to the Page Manager and will thereafter be ignored by the swapper.

SwapIn and SwapOut operations consist of allocating a request packet and issuing the call, in a manner similar to the existing DevIOCall2 interface in OS/2 v1.1. There will be no need to test that the request packet is tiled, or that the function code is correct, etc., so much of the code in DevIOCall2 will not be needed for swap requests and can be avoided.

Since the swap partition cannot be resized while the system is running, all SIOResize (and hence RequestSwapSpace that request an increase) calls will return failure.

File System

This page is intentionally left blank.

Code locking in file system

CODE LOCKING IN FILE SYSTEM

Created: Feb 90 - Ted Miller, Nagarajan Subramanian

Swappable filesystem code will be placed into a number of swappable segments, which are part of the HIGH4CODE group. Each swappable segment will contain one or more groups of code, a group being some number of routines that perform a related task. The segment will be named for its largest component.

Currently, we expect to create approximately 3 segments.

```
1) NMPSWAPCODE: Named pipes:    6553 bytes
                  Fsctl code:     3348 bytes
                  DosFileIO code:  548 bytes
                  OpLock code:     608 bytes
                  DriveLock code:  179 bytes
                  -----
                           11236 bytes total
```

```
2) FATEACODE:          approx. 8K
```

```
3) FSSWAPCODE:         approx 10k
   (FAT OPEN/MKDIR/REN/SEARCH, PIPES, SHUTDOWN,
    NOTIFY, and other infrequently used API workers for FAT)
```

Calling an entry point into a group of code will cause all of the code for that group to be locked. (we don't assume preemptibility for most of the FAT CODE; we want to be safe; and so we have to lock the code down)

For example, a **DosTransactNmPipe** will result in the entire named pipe group (6553) bytes to be locked in. Doing a **DosFileIO** will result in the three routines that comprise the FileIO worker and function router to be locked. The relevant code range will be unlocked upon exit. Since FATEA is in a segment on its own, the full segment is locked.

Each Segment in a group, when they are declared, also has a START label and an END label at the end. This helps us find the length of the segment. For e.g., FSSWAPCODE has _StartFSSWAPCODE and _EndFSSWAPCODE labels automatically generated. (the last seg. in a group will not have an _End Label, because of the way the segm.definition macros are defined. We overcome this by declaring a dummy segment at the end of a group).

The swappable code, when needed, is locked by calling **VMLock**. For the range, we can either use the full segment size itself or the necessary range. We adopt both strategies, for filesystem swapping.

HOW CODELOCKING IS ACCOMPLISHED

A 'code range' is a number of adjacent routines within a swappable segment. Each code range is assigned a group name that will identify it when using code locking macros.

The group name will be equated to an offset into a table of structures. This allows us to avoid a multiply instruction later, when we need to access the data associated with a code lock group. The CodeLock_s is described below.

```
NMPSWAP_GROUP    equ    (0* SIZE CodeLock_s)
FSCTL_GROUP      equ    (1* SIZE CodeLock_s)
FILEIO_GROUP     equ    (2* SIZE CodeLock_s)
OPLOCK_GROUP     equ    (3* SIZE CodeLock_s)
FSSWAP_GROUP     equ    (4* SIZE CodeLock_s)
```

The ranges are physically delineated inside source files by placing labels in them:

```
_StartNMPSWAP_GROUP label byte

    . . .

    <named pipe code>

    . . .

_EndNMPSWAP_GROUP label byte
```

Note that the labels for the start and end addresses of the segments are automatically generated by the build process. This can be used if the parts of a swappable segment are contained in many files, and the whole segment can be locked in at once.

The label names for the start and end of a code range are placed in a table in INITCODE. An Init-time routine is called from fsinit.asm:

```
for(grp=1 to NUMCODELOCKGROUPS) {
    CodeLockArray[grp].startaddr = CodeLockInitInfo[grp].start;
    CodeLockArray[grp].rangelen  = CodeLockInitInfo[grp].end -
                                CodeLockInitInfo[grp].start + 1;
    KSEInit(:CodeLockArray[grp].semaphore, KRNL_MUTEXSEM);
}
```

This routine takes the start and end addresses for each code range, and processes them to form the start addresses and range lengths required by VMLock/Unlock. This data, along with other information, is stored in a CodeLock structure:

```
CodeLock_s      struc
cl_lhand        db        size lockhandle_s dup (?)
cl_count        dw        ?
cl_rangelen     dd        ?
cl_startaddr    dd        ?
cl_semaphore    db        size krnlsem_s dup (?)
CodeLock_s      ends
```

The cl_startaddr and cl_rangelen fields contain the parameters that will be passed to **VMLock**. cl_lhand is the lock handle associated with the code range.

cl_count and cl_semaphore are used to avoid unnecessary calls to **VMLock/Unlock**. The cl_count field counts how many users of the code range currently exist. If, on entry, the count goes from 0 -> 1, a lock is done. If on exit the count goes from 1 -> 0, an unlock is done. The cl_semaphore field guards access to the count field, as VMLock can block and the count could become corrupted if another process entered the code group while the first **VMLock** call were blocked. The semaphore is a Mutex kernel semaphore.

To actually lock a range of code, a macro call is inserted at the beginning of each entry point to the code range, and another at the end of the routine. This macro preserves all registers (including flags). It places the given code group into a register and calls a subroutine that will perform the count checks and locks/unlocks. The subroutine takes its single argument in a register so that it can be called from either 16 or 32 bit code without having to worry about corrupted argvars. If the routine were ever to be called from C, a simple asm interlude would be used, that takes the codegroup off the stack and puts it in the proper register before calling the proper subroutine.

```
CodeLock        macro    group

    CpuMode 386
    pushfd                    ; to be safe
    SaveReg <ds,es,ebx>
    FlatContext <ds,es>,bx,noassume
```

```

        mov     ebx, group
        CALL32  CodeLockProc
        RestoreReg <ebx,es,ds>
        popfd
        CpuMode reset
        ENDM

CodeUnlock      macro    group

        CpuMode 386
        pushfd
        SaveReg <ds,es,ebx>
        FlatContext <ds,es>,bx,noassume
        mov     ebx, group
        CALL32  CodeUnlockProc
        RestoreReg <ebx,es,ds>
        popfd
        CpuMode reset
        ENDM

```

EXAMPLE:

```

Procedure      w_FileIO,HYBRID
        ASSUME  DS:NOTHING,ES:NOTHING,SS:TASKAREA

LocalVar      wfio_cmderror_ptr,DWORD

        CodeLock FILEIO_GROUP          ;***** LOCK DOSFILEIO ROUTINES
        EnterProc <es,di>

        CALLFAR SFFromHandle           ; es:di -> sft entry
        jc      wfio6                  ; if error, go return
        CALLFAR PhysHandle             ; do we have a physical disk handle ?
        jc      wfio6                  ; yes, go return

        . . .

        call    es:[di].FS_FILEIO      ; call remote FSD
        or      ax,ax                  ; any error
        jz      wfio5                  ; no, return with carry clear
        stc                             ; yes, set carry flag

wfio5:  LeaveProc
        CodeUnlock FILEIO_GROUP        ;***** UNLOCK DOSFILEIO ROUTINES
        ret

EndProc w_FileIO

```

Note that this approach requires minor reworking of any routines that need to lock/unlock and have multiple exit points. The overhead for the CodeUnlock macro is not huge, but invoking it more than once per routine should be avoided. We may add support to enforce a one-to-one correspondence between invocations of the CodeLock and CodeUnlock macros, in the style of EnterProc/LeaveProc.

The subroutines that are called by the macros, CodeLockProc and CodeUnlockProc, are 32-bit routines, but not C-callable, as mentioned above.

CodeLockProc:

```

pLockInfo = lockgroup + pCodeLockArray; //address lock info for group
KSEMRequestMutex(:pLockInfo->semaphore);
pLockInfo->count++;
if((pLockInfo->count) == 1) {           //range not already locked
    start = pLockInfo->startaddr;
    size = pLockInfo->rangelen;
    pLockhandle = :(pLockInfo->lockhandle);
    rc = VMLock(start,size,PL_NOBLOCK|PL_LONG,pLockhandle);
    if(rc) Panic;
}
KSEMReleaseMutex(:pLockInfo->semaphore);

```

CodeUnlockProc:

```

pLockInfo = lockgroup + pCodeLockArray; //address lock info for group
KSEMRequestMutex(:pLockInfo->semaphore);
pLockInfo->count--;
if(!(pLockInfo->count)) {               //range needs to be unlocked
    rc = VMUnlock(:pLockInfo->lockhandle);
    if(rc) Panic;
}

```

```

}
KSEMRReleaseMutex(:pLockInfo->semaphore);

```

To get the maximum benefit out of swapping, we have moved some of the HIGH2CODE into HIGH4CODE. HIGH2CODE has only resident code, but HIGH4 now has a resident portion as well as swappable portion. (HIGH2 is nearly full, as we speak, but after we are through, the situation will be improved).

The code locking for FAT Findnotify is done differently. For **findNotifyfirst**, we lock whenever we get a successful **findNotifyfirst** request. We unlock when we do a FAT **FindNotifyClose**. This way we will avoid too many calls to VMlock. (The notify wake procedures that FAT keeps calling from other APIs, are moved to resident code to avoid unnecessary paging. These procedures now check whether notify is in effect, and if so call the actual wake procedure in the swappable portion).

July, 90 - Nagara

The current codelock groups are:

FATEA_GROUP	- Fat EA related code	(fatea.asm extattr.asm)
NMPIPE_GROUP	- Named pipe code	(nampipe?.asm)
FSCTL_GROUP	- FSctl, FSAttach, QFSAttach	(FSCTL.asm)
FILEIO_GROUP	- DOSFileIO	(fhandle.asm)
OPLOCK_GROUP	- OpLock for server	(oplock.asm)
DRVLOCK_GROUP	- Drive Lock/Unlock code	(lockdrv.asm)
NEWNODE_GROUP	- DosMkDir, CreateFile, Rename...	(mknode.asm)
FSSWAP_GROUP	- QPathInfo, DosDelete etc..	(fatdirop.asm)
NOTIFY_GROUP	- FindNotify	(notify.asm)

FindFirst/next code is preemptible and so does not need any lock.

IFS Mechanism

This page is intentionally left blank.

Requirements on The IFS Mechanism

The Installable File System (IFS) Mechanism supports the following:

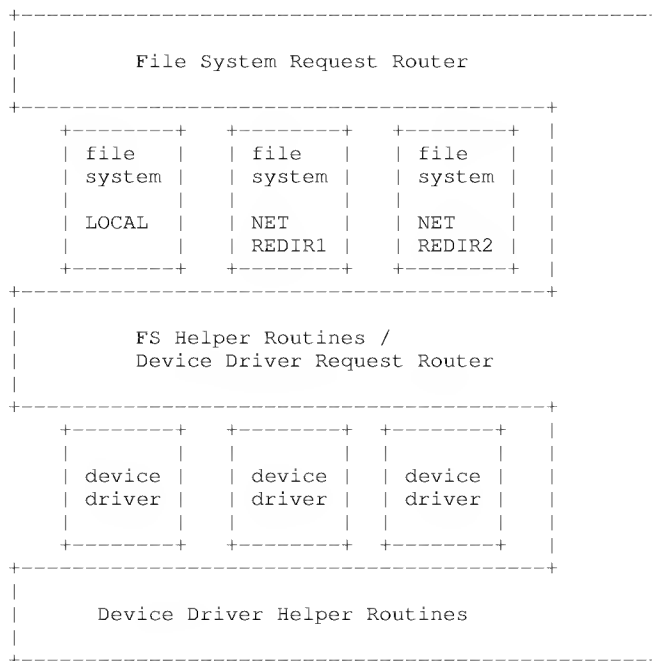
- Coexistence of active file systems in a single PC,
- Multiple logical volumes (partitions),
- Multiple and different storage devices,
- Redirection or connection to remote file systems,
- File system flexibility in managing its data and I/O for optimal performance,
- Transparency at both the user and application level.
- Standard set of File I/O API,
- Existing logical file and directory structure,
- Existing naming conventions,
- File system doing its own buffer management,
- File system doing file I/O without intermediate buffering,
- Extensions to the Standard File I/O API (File System CTL),
- Extensions to the existing naming conventions,
- IOCTL type of communication between a file system and a device driver.

Description

This page is intentionally left blank.

System Relationships

The IFS Mechanism defines the relationships among the operating system, the file systems, and the device drivers. The basic model of the system is represented in the figure.



System Relationships for Installable File Systems

The Request Router directs File I/O system calls to the appropriate file system for processing.

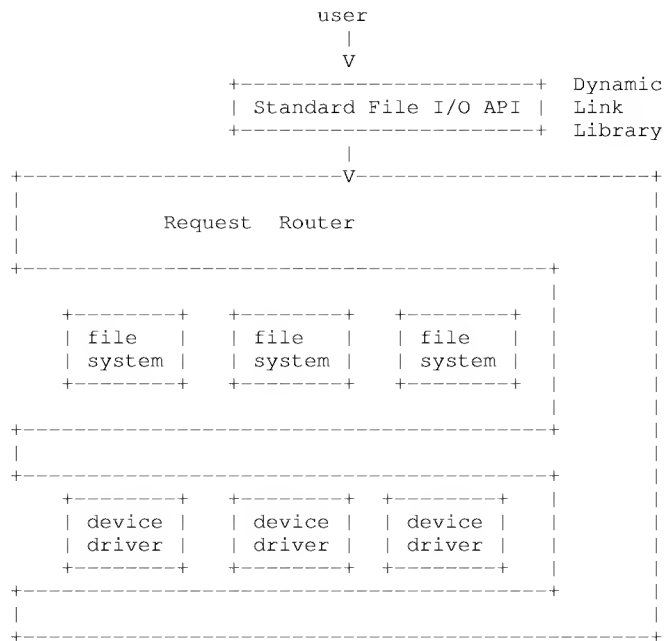
The file systems manage file I/O and control the format of information on the storage media. An installable file system will be referred to as a file system driver or FSD.

The FS Helper Routines provide a variety of services to the file systems.

The device drivers manage physical I/O with devices. Device drivers do not understand the format of information on the media.

Standard File I/O

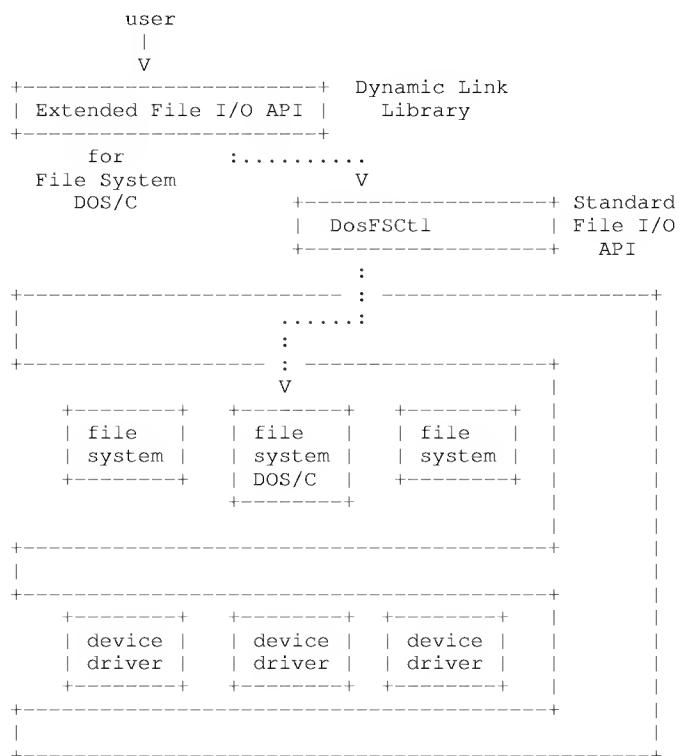
Standard file I/O is performed through the Standard File I/O API. The user makes a system call, and the Request Router passes the request to the correct file system for processing.



Standard File I/O

Extended File I/O

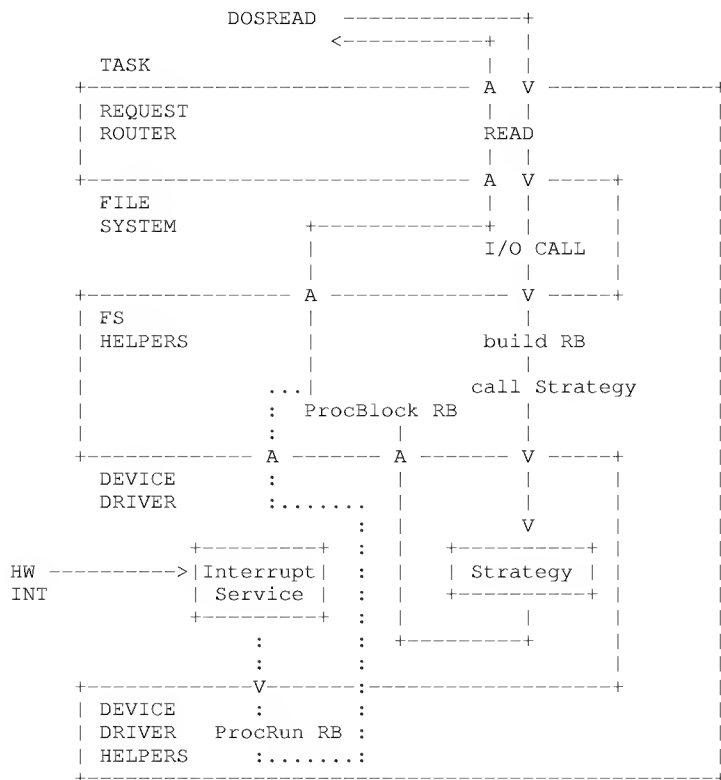
New API may be provided by a file system to implement functions specific to the file system or not supplied through the standard file I/O interface. Any new API would be provided as a dynamic link package that would use the **DOSFCTL** to communicate with the specific file system.



Synchronous vs. Asynchronous I/O

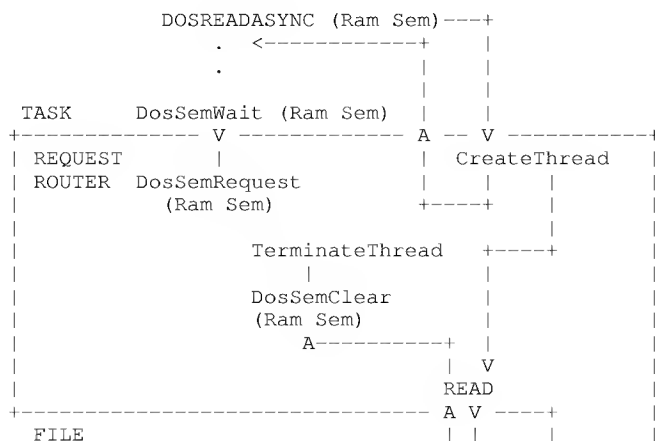
There are two general classes of file I/O that must be taken into account by the IFS Mechanism. The two general classes of file I/O are synchronous I/O and asynchronous I/O.

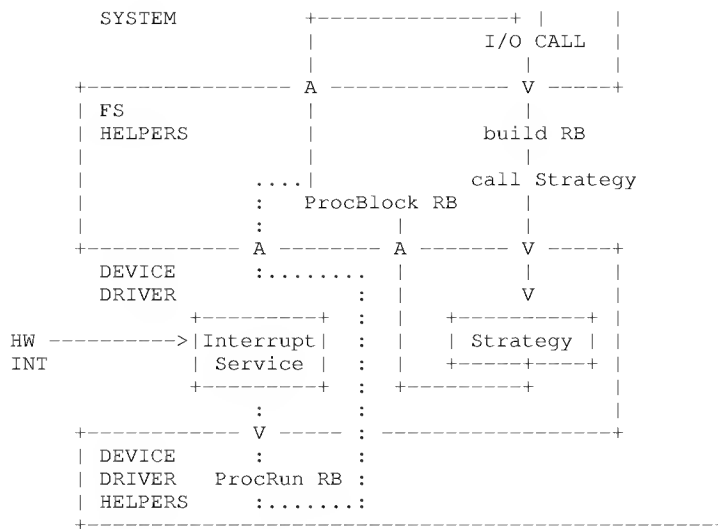
In the case of synchronous file I/O, a task issues a system call for I/O and will not regain control until the operation has been performed. The flow through the model of system relationships is represented in the figure.



Synchronous I/O Example

In the case of asynchronous file I/O, a task issues a system call for I/O and regains control before the I/O has been performed. The flow through the model of system relationships is represented in the figure for asynchronous I/O.





Asynchronous I/O Example

As can be seen by the examples, the file system does not have special knowledge of the nature of the task's original request. One reason for hiding the real nature of a task's request from the file system is to retain control in the kernel. This gives the kernel the freedom to later optimize the handling of asynchronous I/O.

Buffer Management

The IFS mechanism allows an FSD to do its own buffer management.

The FSD moves all data, requiring partial sector I/O, between the application's buffers and its cache buffers. The FS helper routines initiate the I/O for local file systems.

An FSD may use the OS/2-managed buffer cache using FS helpers.

An FSD may do all buffering entirely on its own.

Volume Management

Volume management, i.e., detecting when the wrong volume is mounted and notifying the operator to take corrective action, is handled directly through OS/2 and the device driver. Each FSD is responsible for generating a volume label and 32-bit volume serial number. It is strongly suggested that these be stored in a reserved location in logical sector zero at format time.

It is not required that an FSD use a particular format to store this information. OS/2 will call the FSD to perform operations that might involve it. The FSD is required to update the VPB whenever the volume label or serial number is changed.

When the FSD passes an I/O request to an FS helper routine the driver passes the 32-bit volume serial number and the user's volume label (via the VPB). When the I/O is performed, OS/2 compares the requested volume serial number with the current volume serial number it maintains for the device. This is an in-storage test (no I/O required) performed by checking the Drive Parameter Block's (DPB) VPB of volume mounted in the drive. If unequal, OS/2 signals the critical error handler to prompt the user to insert the volume having the serial number and label specified.

When OS/2 detects a media change in a drive, or the first time a drive is accessed on behalf of an API function call, it will determine the FSD (file system driver) that will be responsible for managing I/O to that volume. OS/2 will allocate a VPB (volume parameter block) and poll the installed FSDs (by calling the FS_MOUNT entry point) until an FSD indicates that it does recognize the media.

The FAT FSD will be the last in the list of FSDs and, by recognizing all media, will act as the default FSD when no other FSD recognition takes place.

Connectivity

There are two classes of file system drivers:

- an FSD which uses a block device driver to do I/O to a local or remote (virtual disk) device. This is called a **local file system**.
- an FSD which accesses a remote system without a block device driver This is called a **remote file system**.

The connection between a drive letter and a **remote file system** is achieved through a command interface.

The connection between a pseudo-character device and a **local or remote file system** is achieved through a command interface.

When a local volume is first referenced OS/2 sequentially asks each FSD in the FSD chain to accept the media, via call to each FSD's **FS_MOUNT** entry point. If no FSD accepts the media then it is assigned to the default FAT file system. Any further attempt made to access an unrecognized media other than by FORMAT, will result in an 'Invalid media format' message.

Once a volume has been recognized, the relationship between drive, FSD, volume serial number, and volume label is remembered. The volume serial number and label are stored in the **Volume Parameter Block**, (VPB). The VPB is maintained by OS/2 for open files (file-handle based I/O), searches, and buffer references. The VPB represents the media.

Subsequent requests for a removed volume require polling the installed FSDs for volume recognition by calling FS_MOUNT. The volume serial number and volume label of the VPB returned by the recognizing FSD and the existing VPB are compared. If the test succeeds, the FSD is given access to the volume. If the test fails, OS/2 signals the critical error handler to prompt the user for the correct volume.

The connection between media and VPB is remembered until all open files on the volume are closed, search references and cache buffer references are removed. Only volume changes cause a re-determination of the media at the time of next access.

IPL Mechanism

A primary DOS disk partition (type 1, 4, or 6) may be used to boot the system. The code for FSDs may reside in any partition readable by a previously installed FSD.

The OS/2 boot partition includes the following: Boot record, basic file system structure, BIOS and DOS files, OS/2 initialization files, ABIOS patch files, CONFIG.SYS, and base device drivers. Boot partitions always contain FAT file systems.

If the media isn't bootable, there need not be a FAT partition present.

Device drivers and FSDs are loaded in the order they appear, and are considered elements of the same ordered set. Thus both device drivers and FSDs may be loaded from installed file systems, so long as they are started in the proper order. For example:

```
DEVICE = c:\devices\diskdriv.sys
REM Block device D: is now defined. (diskdriv.sys controls this.)
IFS = c:\fsd\newfs1.fsd
REM If we assume that D: contains a fixed newfs1 type partition,
REM then we're now ready to use D: to load the device driver and
REM FSD for E:.
DEVICE = d:\root\dev\special.dev
REM Block device E: is now defined.
IFS = d:\root\fsd\special.fsd
REM E: can now be read.
DEVICE = e:\music
```

OS/2 Partition Access

Access to the OS/2 partition on a bootable, logically partitioned media is through the full OS/2 function set. A detailed description of the disk

partitioning design is available in the OS/2 1.1 FPFS.

Permissions

An architecture for secure file systems has been considered but not formalized. There are no secure file system clients identified for the first release of OS/2 incorporating the IFS architecture.

File Naming Conventions

OS/2 1.3 will view 'path names' as ASCII strings and not restrict FSDs to the DOS file name format: xxxxxxxx[.yyy] (8.3 format). This allows an FSD to use whatever naming convention is appropriate for it today and to extend that naming convention in future releases as necessary. DOSQSYSINFO should be called at initialization time to determine maximum path length.

'\ ' and '/' are not valid file name characters. Names in paths are delimited by '\ ' or '/' and have no drive letters. This supports other Operating System name formats (e.g., VM, UNIX, RPS, EDX).

The file names '.' and '..' are reserved for use by hierarchical directory structures for the current directory and the parent of the current directory respectively. The names '.' and '..' receive special processing during canonicalization, and will never be seen by any FSD.

The '.' and '..' notation must be used by file systems using hierarchical directory structures, with the semantics described above. In addition, '.' and '..' must be physically present in all directories.

File system drivers which support hierarchical directory structures must use '\ ' and '/' as path name component separators. '\ ' and '/' are identical in meaning. No other character may be accepted as a path separator. File system drivers which do not support hierarchical directory structures must reject as illegal any use of '\ ' or '/' in path names.

If an FSD uses a component separator within a filename, it must be '\ '. There are no restrictions on the number of components which may be allowed within a file name.

FSD filenames are restricted to the OS/2 character set.

For compatibility reasons, the OS/2 FAT file system will continue to only support the old DOS file name format: xxxxxxxx[.yyy] (8.3 format). It is required that OS/2 and PC/DOS media be compatible within the FAT File System context.

Compatibility with existing DOS 3 applications requires all FSDs to support a superset of the FAT file system's 8.3 format.

IFS requests issued from the 3x box will have the 8.3 truncation rules applied before the FSD receives the request. FSDs do not need to know if a particular request is issued by the 3x box.

Applications are not expected to understand FSD-specific naming conventions. Parsing of names is to only take into account '\ ', '/', '.', and '..'.

After the OS/2 name processing has completed, the path is passed to the FSD using the FS_PROCESSNAME interface for enforcement of FSD-specific naming conventions.

Meta Character Processing

Meta characters will work in the following way for all programs regardless of level number.

Meta characters have two sets of semantics. The first is as search characters, where they are used to select which filenames are returned to the user. The second is as edit characters, used to construct a new name given a source name and a target name specification. The second use occurs in places like "copy *.txt *.old", and is not actually supported anywhere in the kernel except in FCB_rename, and some special FCB editing calls.

Thus, both "*" and "?" have two rules, one for searching, one for copy-editing used in "ren <name-with-metas> <name-with-metas>", and such.

"." has no special meaning itself in searching, but ? gives it one.

"." has a special meaning for editing.

Searching

* matches 0 or more characters, any character, including blank.
It will not cross NUL or \. (Which means it only matches a filename,
not an entire path.)

? matches 0 characters, unless it's "." or the terminating null,
in which case "?" matches 0 characters. It will not cross \.

Any character other than * and ? matches itself, including .

Editing

metas in the source simply match files, and behave just like
any other search meta. (see above)

metas in the target are copy-edit commands, and work like this:

? copies one character, unless what it would copy is a ".",
in which case it copies 0. It also copies 0 characters if
we're at the end of the source string.

* copies characters from the source to the target until it finds
a source character that matches the character following it in
the target.

All non-meta characters write themselves into the target string.

. in the target "syncs" pointers. It causes the source
pointer to match the corresponding "." in the target. They
count from the left.

The DOSEEDITNAME API performs the described operation.

Family API issues

Since the IFS mechanism is neither present in previous releases of OS/2 nor present in real-mode DOS versions 2.0 through 3.3, FAPI will not be extended to support the new interfaces.

FS Utility Support

Each FSD is required to provide a single executable module in order to support the OS/2 FORMAT, CHKDSK, and RECOVER utilities. The FS-provided executable will be invoked by these utilities when performing a FORMAT, CHKDSK, or RECOVER function for that file system. The command line that was passed to the utility will be passed unchanged to the FS-specific executable.

FSD Pseudo-Character Device Support

A pseudo-character device (single file device) may be redirected to an FSD. The behavior of this file is very similar to the behavior of a normal DOS character device. It may be read from (**DosRead**) and written to (**DosWrite**). The difference is that the **DosChgFilePtr** and

DosFileLocks functions can also be applied to the file. The user would perceive this file as a device name for a non-existing device. This file is seen as a character device because the current drive and directory have no effect on the name. This is what happens in DOS today for character devices. For example:

```
FILESYS HOST BCRVMPC1 USERID PASSWORD
```

would define a file (or device) through which host communication can be established. A **TYPE HOST** command would display the latest host created data and a **COPY CON HOST** command could be used to send commands to the host. This example assumes the FSD **BCRVMPC1** can perform the necessary host communication and translation.

The format of an OS/2 pseudo-character device name (i.e., single file device) is that of an ASCIIZ string in the format of a OS/2 filename in a subdirectory called **\DEV**. The pseudo device name **HOST** is accessible at the API level (**DosQFsAttach**) through the path name: **'\DEV\HOST'**.

Extended attributes

Extended attributes (EAs) are a mechanism whereby an application can attach information to a file system object (directories or files) describing the object to another application, to the operating system, or to the FSD managing that object.

This data is not kept as part of the object's data itself; to do this would require all applications to understand the object's format and would give the applications direct access to all other EAs. Normally, applications will deal with a subset of the EAs that may exist on an object.

Each EA has two parts: a name and a value. The name is ascii text that is used to identify a particular EA. There is no other mechanism to identify a particular EA. The name portion of EAs is chosen by the application programmer. This poses the problem of name collision. To address this, a naming convention is adopted. This convention is quite simple; the name is prefixed with the name of the company (or suitable abbreviation) and the name of the product (or suitable abbreviation) that defines the attribute.

All kernel-defined EAs will be prefixed with "DOS."

EA names are restricted to the same character set as filenames.

The value portion of an EA may be arbitrary data or it may be ascii data. There are many different representations for the value portion of EAs. Rather than mandate a specific format, the following guidelines should be used:

If a simplification/optimization can be achieved in the application by representing the value as binary data (such as icons or timestamps) then the representation should be binary.

Otherwise, the representation should be displayable in all character sets (i.e. restricted to characters 0x20 through 0x7E).

EAs may be viewed as a property list attached to objects. The services for manipulating EAs are: add/replace a series of name/value pairs, return name/value pairs given a list of names, and return the total set of EAs.

There are two formats for EAs as passed to OS/2 v1.2 API: Full EAs (FEA) or Get EAs (GEA).

FEAs are complete name/value pairs. In order to simplify and speed up scanning and processing of these names, they are represented as length- preceded data. FEAs are defined as follows:

```
struct FEA {
    unsigned char  reserved;      /* must be 0          */
    unsigned char  cbName;        /* length of name     */
    unsigned long  cbValue;       /* length of value    */
    unsigned char  szName[];      /* asciiz name        */
    unsigned char  aValue[];      /* free-format value  */
};
```

The name length does not include the trailing NUL. The characters that form the name are legal filename characters.

A list of FEAs is a packed set of FEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```

struct FEAList {
    unsigned short cbList;          /* length of list */
    struct FEA list[];              /* packed set of FEA */
};

```

FEA lists are used for adding, deleting, or changing EAs. A particular FSD may store the EAs in whatever format it desires. Certain EAs may be stored to optimize retrieval.

A GEA is an attribute name. Its format is:

```

struct GEA {
    unsigned char cbName;           /* length of name */
    unsigned char szName[];         /* asciiz name */
};

```

The name length does not include the trailing NUL.

A list of GEAs is a packed set of GEA structures preceded by a length of the list (including the length itself) as indicated in the following structure:

```

struct GEAList {
    unsigned long cbList;           /* length of list */
    struct GEA list[];              /* packed set of GEA */
};

```

GEA lists are used for retrieving the values for a particular set of attributes. It is used as input only.

Name lengths of 0 are illegal and are considered in error. A value length of 0 has special meaning. Setting an EA with value length of 0 will cause that attribute to be deleted (if possible). Upon retrieval, a value length of 0 indicates that the attribute is not present.

Setting attributes contained in an FEA list treats the entire FEA list as atomic: all of the changes specified in the list are applied or none will. This is required since the attributes together may specify some behaviour for the file system and may not make sense being set independently.

The routing of EA requests is accomplished via the IFS routing mechanism. EA requests that apply to names are routed to the remote FSD attached to the specified drive or to the local FSD managing the volume in the named drive. Those requests that apply to a handle (file or directory) are routed to the FSD attached to the handle. No interpretation of either FEA lists nor GEA lists is performed by the IFS router. It is the responsibility of each FSD to provide support for EAs. It is expected that some FSDs will be unable to store EAs (e.g. UNIX- and MVS-compatible file systems)

The FAT FSD implementation will provide for the complete implementation of EAs. There will be no special EAs for the FAT FSD.

All EA manipulation is performed using the following structure; the relevance of each field is described within each API:

```

struct EAOP {
    struct GEAList far * fpGEAList; /* GEA set */
    struct FEAList far * fpFEAList; /* FEA set */
    unsigned short offError;        /* offset of FEA err */
};

```

FSD File Image

An FSD loads from a file which is in the format of a standard OS/2 dynamic link library file. Exactly one FSD resides in each file. The FSD exports information to OS/2 using a set of predefined public names.

The FSD is initialized by a call to the exported entry point FS_INIT.

FS entry points for Mount, Read, Write, etc. are exported with known names as standard far entry points.

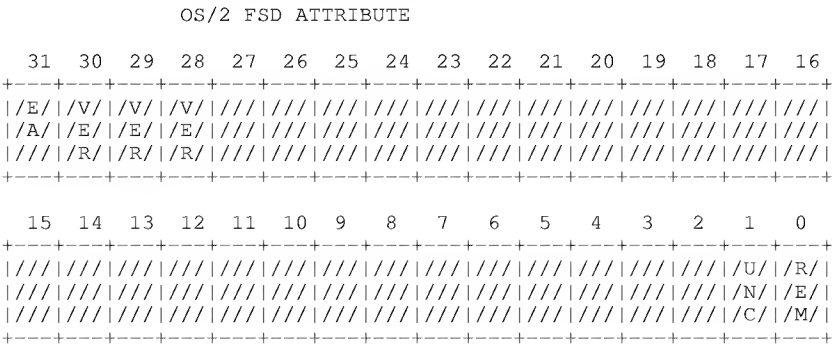
The FSD exports its name as a public asciiz character string under the name 'FS_NAME'. All comparisons with user specified strings will be done similar to file names; case is ignored and embedded spaces are significant. FS_NAMES, however, may be input to applications by users.

Embedded spaces should be avoided. The name exported as FS_NAME need NOT be the same as the 1-8 byte FSD name in the boot sector of formatted media, although it may be. The ONLY name the kernel pays any attention to whatsoever, and the only name accessible to user programs via the API is the name exported as FS_NAME.

In addition to various entry points, the FSD must export a dword bit vector of attributes. Attributes are exported under the name 'FS_ATTRIBUTE'. FS_ATTRIBUTE specifies special properties of the FSD and is described in the next section.

FSD Attribute, FS_ATTRIBUTE

The format of the OS/2 FS_ATTRIBUTE field is defined below:



<u>BITS</u>	DESCRIPTION
<u>31</u>	FSD Extended Attributes. If 1, FSD has extended attributes. If 0, FS_ATTRIBUTE is only FSD attribute information.
<u>30-28</u>	VERSION NUMBER - FSD version number.
<u>27-2</u>	RESERVED.
<u>1</u>	UNC - Universal Naming Convention bit. Set if FSD supports Universal Naming Convention. If set, then FSD represents the default network redirector. Only one FSD may have this bit set.
<u>0</u>	REM - remote file system. Clear if a local FSD, Set if a remote FSD. Local FSDs always use dynamic media attachment. Remote FSD's always use static media attachment. No FSD supports both static and dynamic media attachment.

FSD Initialization

FSD initialization occurs at system initialization time. FSDs are loaded via the **IFS =** configuration command in **CONFIG.SYS**. Once the FSD has been loaded, the FSD's initialization entry point is called to initialize it.

FSDs are structured the same as dynamic link library modules. Once an FSD is loaded, the initialization routine FS_INIT is called. This gives

the FSD the ability to process any parameters that may appear on the CONFIG.SYS command line, which are passed as a parameter to the FS_INIT routine. A LIBINIT routine in an FSD will be ignored.

OS/2 FSDs initialize in protect mode. Because of the special state of the system, an FSD may make dynamic link system calls at init-time. The list of system calls that an FSD may make are as follows:

```
FSD_INIT
DYNAMIC LINK SYSTEM CALLS
DosBeep
DosChgFilePtr
DosClose
DosDelete
DosDevConfig
DosDevIoCtl
DosFindClose
DosFindFirst
DosFindNext
DosFreeSeg
DosGetEnv
DosGetInfoSeg
DosGetMessage
DosOpen
DosPutMessage
DosQCurDir
DosQCurDisk
DosQFileInfo
DosQFileMode
DosQSysInfo
DosRead
DosReallocSeg
DosWrite
```

To release code and data needed only by the FSD's initialization routine, the FSD simply applies DosFreeSeg or DosReallocSeg to the appropriate segments.

It should be noted that multiple code and data segments are not discarded by the loader as in the case of device drivers.

The FSD may call DosGetInfoSeg to obtain access to the global and process local information segments. The local segment may be used in the context of all processes without further effort to make it accessible and has the same selector. The local infoseg is not valid in real mode or at interrupt time.

OS/2 and PC/DOS 3.4 Boot Record and BIOS Parameter Block

The extended Boot Record structure:

```
Extended_Boot  struc
Boot_jump      db      3 dup (?)
Boot_OEM       db      8 dup (?)
Boot_BPB       db      (size Extended_BPB) dup (?)
Boot_DriveNumber db      ?
Boot_CurrentHead db      ?
Boot_Sig       db      41      ;Indicate Extended Boot structure.
Boot_Serial    dd      ?
Boot_Vol_Label db      11 dup (?)
Boot_System_ID db      8 dup (?); "FAT      ", "OTHER_FS"
Extended_Boot  ends
Area for code and messages
Boot_Signature db      55h, 0AAh
```

Where

Serial is the 32-bit binary volume serial number for the media.

System_ID is an 8-byte name written when the media is formatted. It is used by FSDs to identify their media, but need NOT be the same as the name the FSD exports via FS_NAME, and is NOT the name users employ to refer to the FSD. (They may, however, be the same names.)

Vol_Label is the 11-byte ASCII label of the disk/diskette volume. FAT filesystems must ALWAYS use the volume label in the boot sector for compatibility reasons. A FSD may use the one in the standard boot sector if it so desires.

The extended BPB structure is a super-set of the conventional BPB structure.

```
Extended_BPB  struc
BytePerSector    dw    ?
SectorsPerCluster db    ?    ;See the remark below
ReservedSectors  dw    ?
NumberOfFATs     db    ?    ;0 for Non Fat system
RootEntries      dw    ?    ;See the remark below
TotalSectors     dw    ?    ;if 0 then use Ext_TotalSectors
MediaDescriptor  db    ?    ;See the remark below
SectorsPerFat    dw    ?    ;See the remark below
SectorsPerTrack  dw    ?
Heads            dw    ?
HiddenSectors    dd    ?    ;32 bit hidden sectors
Ext_TotalSectors dd    ?    ;32 bit total sectors
Extended_BPB  ends
```

Any Non Fat-based FSD can ignore, or use the fields of SectorsPerCluster, RootEntries, and SectorsPerFat entries for its own purposes, if necessary.

Although, nothing can prevent a media smaller than 32MB from using the Ext_TotalSectors, instead of TotalSectors. For compatibility reasons, IBM FORMAT will use TotalSectors for media =< 32MB.

IFS Commands

This page is intentionally left blank.

FILESYS User Command

Purpose

The **FILESYS** Command specifies a drive (on a network computer) or an FSD-defined device name that you want to start or stop using with a specific installable file system driver (FSD).

Syntax

FILESYS { [d: | device] [FSDname {parms} | /D] }

Format 1

Attaching a drive to a network FSD

FILESYS d: FSDname {parms}

Format 2

Attaching a reserved device name to an FSD

FILESYS device FSDname {parms}

Format 3

Detaching a drive from a network FSD

FILESYS d: /D

Format 4

Detaching a reserved device name from an FSD

FILESYS device /D

Format 5

Displaying drive/device status

FILESYS

Where

FSDname is the name FSD to attach to the drive. An FSD name is defined by the FS_NAME export from the FSD. It is an ASCIIZ name.

{parms} specifies zero or more single strings to pass to the FSD. To a network FSD (a redirector) this would include the network path, password, and any other text required by the referenced FSD.

/D specifies to break or disconnect the attachment of a drive or an FSD pseudo device name.

Remarks

Formatted media (or formatted extended partition volume of a media) has a 1-8 byte FSD name which is determined by the FSD, but need not be the same as the FSD name used for calling the FSD. The name on the media is used solely by FSDs to recognize media which belongs to them. The default name is FAT.

IFS CONFIG.SYS Function

Purpose

The **IFS** command starts (loads and initializes) an FSD.

Format

IFS = {drive:}{path}name{.ext} {parms}

Where

{drive:}{path}name{.ext} specifies the FSD to load and initialize.

parms represents an FSD-defined string of initialization parameters.

IFS API Function Calls

File I/O Function Call Summary

DosBufReset

Flush file buffers.

DosChDir

Change current directory.

DosChgFilePtr

Move the file read/write pointer.

<u>DosClose</u>	Close file handle.
<u>DosDelete</u>	Delete file.
<u>DosDevIoCtl</u>	I/O control for devices.
<u>DosDupHandle</u>	Duplicate file handle.
<u>DosEditName</u>	Transform source string using editing string
<u>DosFileIO</u>	Multi-function call.
<u>DosFileLocks</u>	Lock/unlock a range of bytes in an opened file.
<u>DosFindClose</u>	Terminate usage of a directory search handle.
<u>DosFindFirst</u>	Find first matching file.
<u>DosFindNext</u>	Find next matching file.
<u>DosFindNotifyClose</u>	Terminate usage of a find-notify handle.
<u>DosFindNotifyFirst</u>	Start monitoring directory for changes.
<u>DosFindNotifyNext</u>	Continue monitoring directory for changes.
<u>DosFsAttach</u>	Attach a drive or device to an FSD.
<u>DosFsCtl</u>	File system control
<u>DosMkDir</u>	Make subdirectory.
<u>DosMove</u>	Move a file or subdirectory.
<u>DosNewSize</u>	Change size of a file.
<u>DosOpen</u>	Open/create a file.
<u>DosQCurDir</u>	Query current directory.
<u>DosQCurDisk</u>	Query current default drive.
<u>DosQFHandState</u>	Query file handle state information.
<u>DosQFileInfo</u>	Query file information.
<u>DosQFileMode</u>	Query file mode.
<u>DosQFsAttach</u>	Query attached FSD information.
<u>DosQFsInfo</u>	

<u>DosQHandType</u>	Query file system information.
<u>DosQPathInfo</u>	Query handle type.
<u>DosQSysInfo</u>	Query path information.
<u>DosQVerify</u>	Query system Information
<u>DosRead</u>	Query the verify setting.
<u>DosReadAsync</u>	Read from a file.
<u>DosRmdir</u>	Async Read from a file.
<u>DosScanEnv</u>	Remove subdirectory.
<u>DosSearchPath</u>	Scan environment.
<u>DosSelectDisk</u>	Search path.
<u>DosSetFileInfo</u>	Select disk.
<u>DosSetFileMode</u>	Set file information.
<u>DosSetFHandState</u>	Set file mode.
<u>DosSetFsInfo</u>	Set file handle state.
<u>DosSetMaxFH</u>	Set file system information.
<u>DosSetPathInfo</u>	Define new maximum file handle.
<u>DosSetVerify</u>	Set path information.
<u>DosWrite</u>	Set verify setting.
<u>DosWriteAsync</u>	Write to a file or device.
	Async write to a file or device.

Application File I/O Notes

File handle values of 0xFFFF do not represent actual file handles but are used through-out the file system interface to indicate specific actions to be taken by the file system. Usage of this 'special file handle' where it is not expected by the file system will result in an error.

Null pointers are defined to be **0x00000000** throughout this specification.

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, may not necessarily support all the described information kept on a file basis. When such is the case, FSDs are required to return to the application a null (zero) value for the unsupported parameter.

An FSD may support version levels of files.

Dates have the following format:

Date bit	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
mapping	y	y	y	y	y	y	y	m	m	m	m	d	d	d	d	d

Where:

mm is the month (1-12)

dd is the day of month (1-31)

yy is number of years since 1980

Times have the following format:

Time bit	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
mapping	h	h	h	h	h	m	m	m	m	m	m	x	x	x	x	x

Where:

hh is the binary number of hours (0-23)

mm is the binary number of minutes (0-59)

xx is the binary number of two-second increments

DD Errors

Some API below may return device-driver/device-manager generated errors. Rather than listing the errors with each API, these error codes are:

- ERROR_WRITE_PROTECT - the diskette in the drive has write-protection enabled.
- ERROR_BAD_UNIT - there is a breakdown of internal consistency between OS/2's mapping between logical drive and device driver. Internal Error.
- ERROR_NOT_READY - the device driver detected that the device is not ready.
- ERROR_BAD_COMMAND - there is a breakdown of internal consistency between OS/2's idea of the capability of a device driver and that device driver.
- ERROR_CRC - the device driver has detected a CRC mismatch.
- ERROR_BAD_LENGTH - there is a breakdown of internal consistency between OS/2's idea of the length of a request packet and the device driver's idea of that length. Internal Error.
- ERROR_SEEK - the device driver detected an error during a seek operation.
- ERROR_NOT_DOS_DISK - the disk is not recognized as being OS/2 manageable.
- ERROR_SECTOR_NOT_FOUND - the device is unable to find the specific sector.
- ERROR_OUT_OF_PAPER - the device driver has detected that the printer is out-of-paper.
- ERROR_WRITE_FAULT - other write-specific error.
- ERROR_READ_FAULT - other read-specific error.

- ERROR_GEN_FAILURE - other error.

There are also errors defined by and specific to the device driver themselves. These are indicated by either 0xFF or 0xFE in the high byte of the error code.

Error codes listed below are not complete as each FSD may generate errors based upon its own circumstances. The errors below are those generated by the IFS router and the FAT file system.

Application File I/O Function Calls

This page is intentionally left blank.

DosBufReset - Commit file's cache buffers

Purpose

Flushes requesting process's cache buffers for the specified file handle or for all file handles attached to that process.

Format

Calling Sequence:

```
EXTRN DosBufReset:FAR

PUSH WORD FileHandle ; File handle
CALL DosBufReset
```

Where

FileHandle is the file handle whose buffers are to be flushed. If **FileHandle** == 0xFFFF, as above except that all file handles attached to the process are flushed.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE if the specified file handle is invalid or the handle is attached to a physical device.
- Device-driver/device-manager [errors listed](#).

Remarks

The file's directory entry is updated, (as if the file had been closed using **DosClose**), but the file remains in the open state.

The application should be aware that usage of this function to flush all of the requesting process's cache buffers could have the undesirable effect of requiring the user to mount and unmount a large number of removable volumes.

Named Pipe Considerations

Performs an operation analogous to the defined one of forcing the buffer cache to disk. For named pipes,

DosBufReset blocks the caller until all data written by the caller has been successfully read by the other end. This allows communicating processes to synchronize their dialogs.

DosChDir - Change The Current Directory

Purpose

Define the current directory for the requesting process.

Format

Calling Sequence:

EXTRN DosChDir:FAR

```
PUSH@ ASCIIZ DirName ; Directory path name
PUSH  DWORD 0         ; Reserved (must be zero)
CALL  DosChDir
```

Where

DirName is the ASCIIZ directory path name.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_FILENAME_EXCED_RANGE - the directory path specified is unacceptable to the FSD managing the volume.
- ERROR_PATH_NOT_FOUND - the drive letter is invalid, there are too many .., there are wildcards in the directory path, or a component of the directory path is not present.
- ERROR_NOT_ENOUGH_MEMORY - unable to allocate storage for current directory
- ERROR_DRIVE_LOCKED - the drive is locked by another process
- ERROR_INVALID_PARAMETER - the DWORD reserved field is not 0.
- Device-driver/device-manager [errors listed](#).

Remarks

If any member of the directory path does not exist, then the directory path is not changed. The current directory is changed only for the requesting process.

DosChgFilePtr - Change File Read Write Pointer

Purpose

Moves the read/write pointer according to the method specified.

Format

Calling Sequence:

EXTRN DosChgFilePtr:FAR

```
PUSH  WORD  FileHandle ; File handle
PUSH  DWORD Distance   ; Distance to move in bytes
```

```

PUSH WORD MoveType ; Method of moving (0, 1, 2)
PUSH@ DWORD NewPointer ; New Pointer Location
CALL DosChgFilePtr

```

Where

FileHandle is the file handle.

Distance is the signed distance (offset) to move in bytes.

MoveType is the method of moving (0, 1, 2).

- If value = 0, the pointer is moved **Distance** bytes (offset) from the beginning of the file.
- If value = 1, the pointer is moved to the current location plus offset.
- If value = 2, the pointer is moved to the end-of-file plus offset. This method can be used to determine the file's size.

NewPointer is a double word area where the system returns the **new pointer location**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the specified handle is not valid in the current process, the handle is attached to a physical device
- ERROR_SEEK_ON_DEVICE - the handle is attached to a pipe or character device.
- ERROR_INVALID_FUNCTION - the parameter MoveType is invalid
- ERROR_NEGATIVE_SEEK - the resulting position within the file is negative.
- ERROR_DIRECT_ACCESS_HANDLE - the handle is for a direct open and the resulting position is beyond the physical end of the media.

Remarks

The read/write pointer in a file is a signed 32-bit number.

It is illegal to seek to a negative position in a file.

It is illegal to seek on a character device or pipe.

DosClose - Close a File Handle

Purpose

Closes the specified file handle.

Format

Calling Sequence:

```
EXTRN DosClose:FAR
```

```

PUSH WORD FileHandle ; File Handle
CALL DosClose

```

Where

FileHandle is the handle returned by **DosOpen** or **DosMakePipe**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the specified handle is not open or is attached to a physical device.
- **ERROR_FILE_NOT_FOUND** - the directory entry that is thought to contain the name of the file doesn't.
- Device-driver/device-manager [errors listed](#).

Remarks

Closing a file handle causes the file to be closed, the directory to be updated, and all internal buffers for that file to be written to the media.

Named Pipe Considerations

Closes a pipe by handle. When all handles referencing one end of a pipe are closed the pipe is broken. If the client end closes, no other process can reopen the pipe until the serving end issues a **DosDisconnectNmPipe** followed by a **DosConnectNmPipe**. If the server end closes, the pipe will be deallocated when the last client handle is closed or immediately if the pipe is already broken.

DosDelete - Delete a File

Purpose

Removes a directory entry associated with a filename.

Format

Calling Sequence:

EXTRN DosDelete:FAR

```
PUSH@ ASCIIZ FileName ; Filename path
PUSH  DWORD  0         ; Reserved (must be zero)
CALL  DosDelete
```

Where

FileName is the ASCIIZ file name.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_FILENAME_EXCED_RANGE** - the path specified is unacceptable to the FSD managing the volume.
- **ERROR_PATH_NOT_FOUND** - the drive letter is invalid, there are too many .., there are wildcards present, or a component of the directory path is not present.
- **ERROR_FILE_NOT_FOUND** - the filename at the end of the path is not found.
- **ERROR_ACCESS_DENIED** - the path specified a directory or a device, or the file is read-only.
- **ERROR_SHARING_VIOLATION** - the file specified is currently in use.
- **ERROR_SHARING_BUFFER_EXCEEDED** - there is not enough memory to hold sharing information.
- Device-driver/device-manager [errors listed](#).

Remarks

Wildcard characters are not allowed in any part of the ASCII string. Read-only files cannot be deleted by this call. To delete a read-only file, you must first use function **DosSetFileMode** (Set File Mode) to change the file's read-only attribute to 0, then delete the file.

DosDevIOctl - I/O Control for Devices

Purpose

Perform control functions on the device specified by the file or device handle.

Format

Calling Sequence:

EXTRN DosDevIOctl1:FAR

```
PUSH@ OTHER Data          ; Data area
PUSH@ OTHER ParmList       ; Command arguments
PUSH WORD Function         ; Device function
PUSH WORD Category         ; Device category
PUSH WORD DevHandle        ; Specifies the device
CALL DosDevIOctl1
```

EXTRN DosDevIOctl12:FAR

```
PUSH@ OTHER Data          ; Data area
PUSH WORD DataLength       ; Data area length
PUSH@ OTHER ParmList       ; Command arguments
PUSH WORD ParmListLength   ; Command arg's list length
PUSH WORD Function         ; Device function
PUSH WORD Category         ; Device category
PUSH WORD DevHandle        ; Specifies the device
CALL DosDevIOctl12
```

Where

Data is a data area.

DataLength is the length of the data buffer.

ParmList is a command-specific argument list.

ParmListLength is the length of the command-specific argument list.

Function is the device-specific function code. The valid range is 0 to 255.

Category is the device category. The valid range is 0 to 255.

DevHandle is a device handle returned by **DosOpen**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the handle specified is either not in use, is attached to a physical device and category 9 is not specified or is not attached to a physical device and category 9 is specified.
- **ERROR_INVALID_FUNCTION** - the specified function is illegal for the particular category and handle.

- **ERROR_INVALID_CATEGORY** - the specified category is illegal for the particular function and handle.
- **ERROR_INVALID_DRIVE** - the drive specified in a set-logical-drive-map call is not supported by the device driver.
- **ERROR_INVALID_PARAMETER** - (**DosDevIOCtl2**) a 0 length was specified with a non-null pointer for either Data or ParmList.
- Device-driver/device-manager [errors listed](#).

Remarks

This function provides a generic, expandable IOCTL facility.

A null (zero) value for **Data** specifies that this parameter is not defined for the generic IOCTL function being specified. A null value for **Data** causes the value passed in **DataLength** to be ignored.

A null (zero) value for **ParmList** specifies that this parameter is not defined for the generic IOCTL function being specified. A null value for **ParmList** causes the value passed in **ParmListLength** to be ignored.

The kernel will format a generic IOCTL packet and call the device driver. Since v1.0 and v1.1 device drivers do not understand generic IOCTL packets with DataLength and ParmListLength, the kernel will not pass these fields to the device driver. Device drivers that are marked as being level 2 or higher (see device driver header spec) must support receipt of the generic IOCTL packets with associated length fields.

It is illegal to pass in a non-null pointer with a zero length.

DosDupHandle - Duplicate a File Handle

Purpose

Returns a new file handle for an open file that refers to the same file at the same position.

Format

Calling Sequence:

EXTRN DosDupHandle:FAR

```
PUSH  WORD OldFileHandle ; Existing file handle
PUSH@ WORD NewFileHandle ; New file handle
CALL  DosDupHandle
```

Where

OldFileHandle is the current file handle.

NewFileHandle is a word file handle that is made the duplicate of **OldFileHandle** or is 0xFFFF. A value of 0xFFFF causes the DOS to allocate a new file handle and use that as the destination of the duplication. Otherwise, the function uses **NewFileHandle** as the destination of the duplication.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the OldFileHandle is invalid or is attached to a physical device.
- **ERROR_TOO_MANY_OPEN_FILES** - NewFileHandle is 0xFFFF and all handles for the process are in use.
- **ERROR_INVALID_TARGET_HANDLE** - the specified NewFileHandle is beyond the range allocated to the process, either by default, or by DOSSETMAXFH.

- Device-driver/device-manager [errors listed](#).

Remarks

Duplicating the handle duplicates and ties all handle-specific information between **OldFileHandle** and **NewFileHandle**.

A file handle value other than 0xFFFF specified in **NewFileHandle** causes that file handle to be closed before it is redefined as the duplicate of **OldFileHandle**.

The valid values for **NewFileHandle** include: 0xFFFF, 0x0000 (Standard Input), 0x0001 (Standard Output), 0x0002 (Standard Error), or the handle of any currently open file. Avoid using other arbitrary values.

If you move the read/write pointer of either handle by a **DosRead**, **DosWrite**, or **DosChgFilePtr** function call, the pointer for the other handle is also changed.

Issuing a **DosClose** against a file handle does not affect the duplicate handle.

DosEditName - Transform source string using editing string

Purpose

Transform a source string into a destination string using an editing string and OS/2 meta editing semantics.

Format

Calling Sequence:

```

EXTRN DosEditName:FAR

PUSH WORD EditLevel ; Level of meta editing semantics
PUSH@ ASCIIZ SourceString ; String to transform
PUSH@ ASCIIZ EditString ; Editing string
PUSH@ OTHER TargetBuf ; Destination string buffer
PUSH WORD TargetBufLen ; Destination string buffer length
CALL DosEditName

```

Where

EditLevel is the version of meta editing semantics to use in transforming the source string.

- If **EditLevel** value is 0x0001, then OS/2 version 1.2 editing semantics are used.

SourceString is the ASCIIZ string to transform. It should contain just the component of the pathname to edit, not the entire path.

EditString is the ASCIIZ string to use for editing.

TargetBuf is the buffer to store the resulting ASCIIZ string in.

TargetBufLen is the length of the buffer to store the resulting string in.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_PARAMETER** - the **EditLevel** is invalid or the buffer size is larger than the maximum component length.
- **ERROR_BUFFER_OVERFLOW** - the specified buffer is too small to contain the destination string.

Remarks

Typical uses of this API are copy and rename/move. For a **SourceString** "foo.bar" and an **EditString** "*.baz", the

destination string is "foo.baz" for OS/2 version 1.2 meta editing semantics.

Some characters in the destination string may be uppercased.

DosFileIO - File I/O

Purpose

Perform multiple lock, unlock, seek, read, and write I/O.

Format

Calling Sequence:

EXTRN DosFileIO:FAR

```
PUSH WORD FileHandle      ; File handle
PUSH@ OTHER CommandList   ; Ptr to command buffer
PUSH WORD CommandListLen  ; Length of command buffer
PUSH@ WORD ErrorOffset    ; Ptr to command error offset
CALL DosFileIO
```

Where

FileHandle is the handle of the file of interest.

CommandList is a list that contains entries indicating what commands will be performed. The operations (CmdLock, CmdUnlock, CmdSeek, CmdIO) are performed as atomic operations until all are complete or until one fails. CmdLock allows a multiple range lock to be executed as an atomic operation, CmdUnlock does the same for multiple range unlocks. Each operation will be guaranteed to execute in one piece.

For CmdLock, the command format is:

```
CmdLock Struc
Cmd      dw      0      ; 0 for lock operations
Timeout dd      ?      ; ms timeout for success of all locks
LockCnt  dw      ?      ; Number of locks that follow
CmdLock Ends
```

which is followed by a series of records of the following format:

```
Lock      Struc
Share     dw      ?      ; 0 for exclusive, 1 for read-only access
Start     dd      ?      ; start of locked region
Length    dd      ?      ; length of locked region
Lock      Ends
```

If a lock within a **CmdLock** causes a timeout, none of the other locks within the scope of **CmdLock** are in force since the lock operation is viewed as atomic.

CmdLock.Timeout is the count in milliseconds, until the requesting process is to resume execution if the requested locks are not available. The meaning of the values specified are:

- 0xFFFFFFFF means the process will wait indefinitely until the requested locks become available.
- 0x00000000 means return immediately to the requesting process
- Otherwise, CmdLock.Timeout is the number of milliseconds to wait until the requested locks become available.

Lock.Share defines the type of access other processes may have to the file-range being locked.

- If Lock.Share == 0, other processes have 'No-Access' to the locked range. The current process will have

both read and write access to the locked range. No region with Lock.Share == 0 may overlap with any other locked region.

- If Lock.Share == 1, other processes have 'Read-Only' access to the locked range. The current process will have 'Read-Only' access to the locked range also. A range locked with Lock.Share == 1 may overlap with other ranges locked with Lock.Share == 1, but may not overlap with other ranges locked with Lock.Share == 0.

For CmdUnlock, the command format is:

```
CmdUnlock Struc
Cmd      dw 1      ; 1 for unlock operations
UnlockCnt dw ?      ; Number of unlocks that follow
CmdUnlock Ends
```

which is followed by a series of records of the following format:

```
UnLock Struc
Start  dd ?      ; start of locked region
Length dd ?      ; length of locked region
UnLock Ends
```

For CmdSeek, the command format is:

```
CmdSeek Struc
Cmd      dw ?      ; 2 for seek operation
Method   dw ?      ; 0 for absolute,
                  ; 1 for relative to current,
                  ; 2 for relative to EOF.
Position dd ?      ; file seek position or delta
Actual   dd ?      ; actual position seeked to
CmdSeek  Ends
```

For CmdIO, the command format is:

```
CmdIO Struc
Cmd      dw ?      ; 3 for read, 4 for write
Buffer@  dd ?      ; ptr to the data buffer
BufferLen dd ?      ; number of bytes requested
Actual   dd ?      ; number of bytes actually
                  ; transferred
CmdIO    Ends
```

CommandListLen is the length in bytes of **CommandList**.

ErrorOffset points to where the system stores the byte offset relative to the start of the record of where an error occurred.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_PARAMETER - invalid command
- ERROR_ACCESS_DENIED - the file is not accessible because another process has access to it.
- ERROR_SHARING_BUFFER_EXCEEDED - there is not enough memory to hold sharing information.
- ERROR_INTERRUPT - the current thread received a signal.
- ERROR_SEEK_ON_DEVICE - the handle is attached to a pipe or character device.
- ERROR_NEGATIVE_SEEK - the resulting position within the file is negative.
- ERROR_DIRECT_ACCESS_HANDLE - the specified handle is opened with the direct open bit

set.

- ERROR_INVALID_HANDLE - the file handle specified is not in use or is attached to a physical device.
- ERROR_LOCK_VIOLATION - a lock operation timed out or a read/write operation encountered a lock.
- Device-driver/device-manager [errors listed](#).

Remarks

This function provides a simple mechanism for combining the following operations into a single request and providing improved performance particularly in a networking environment:

- unlocking and locking of multiple file ranges
- changing of the file position pointer
- read and/or write IO

Provides a simple mechanism for excluding other processes read/write or write access to regions of the file. If another process attempts to read or write a 'No-access' region, or attempts to write in a 'Read-only' region, the system call will return an error indicating exclusion to that region. If the time-out occurs before the locking can be completed, the function returns to the caller with an unsuccessful return code.

The caller may return after the timeout period has expired without receiving an ERROR_SEM_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

A range to be locked must first be cleared of any locked subranges or locked overlapping ranges.

DosFileLocks - File Lock Manager

Purpose

Unlock and/or lock a range in an opened file.

Format

Calling Sequence:

EXTRN DosFileLocks:FAR

```
PUSH WORD FileHandle      ; File handle
PUSH@ OTHER UnLockRange   ; Unlock range
PUSH@ OTHER LockRange     ; lock range
CALL  DosFileLocks
```

Where

FileHandle is the file handle.

UnLockRange identifies the range in the file of the region to be unlocked:

```
UnLockRange struc
FileOffset dd      ?    ; offset where unlock begins
RangeLength dd      ?    ; length of region unlocked
UnLockRange ends
```

A null pointer (a dword of zero) to UnLockRange specifies that unlocking is not required."

LockRange identifies the range in the file of the region to be locked:

```
LockRange  struc
```

```
FileOffset dd      ?    ; offset where lock begins
RangeLength dd     ?    ; length of region locked
LockRange ends
```

A null pointer (a dword of zero) to LockRange specifies that locking is not required."

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - handle specified is not open or is attached to a physical device.
- ERROR_LOCK_VIOLATION - the range specified for locking overlapped a previously locked range or the range specified for unlocking did not exactly match a previously locked range.
- ERROR_SHARING_BUFFER_EXCEEDED - there is no more room for locks.

Remarks

This function provides a simple mechanism for unlocking and/or locking a file range.

If unlocking is specified, the function first unlocks the specified area using **UnLockRange**. After **UnLockRange** is processed, then the locking of a range (if specified via **LockRange**), is done.

Closing a file with locks still in force causes the locks to be released in no defined order.

Terminating with a file open and having issued locks on that file causes the file to be closed and the locks to be released in no defined order.

Provides a simple mechanism for excluding other processes read/write access to regions of the file. The locked regions can be anywhere in the logical file. Locking beyond end-of-file is not an error. It is expected that the time in which regions are locked will be short. Duplicating the handle duplicates access to the locked regions. Access to the locked regions is not duplicated across the **DosExecPgm** system call. The proper method for using locks is not to rely on being denied read or write access, but attempting to lock the region desired and examining the error code.

A range to be locked must first be cleared of any locked subranges or locked overlapping ranges.

DosFindClose - Close Find Handle

Purpose

Closes the association between a directory handle and a **DosFindFirst** or **DosFindNext** directory search function.

Format

Calling Sequence:

```
EXTRN DosFindClose:FAR
```

```
PUSH WORD DirHandle ; Directory search handle
CALL DosFindClose
```

Where

DirHandle is the handle previously associated by the system with a **DosFindFirst** or **DosFindNext** directory search function.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the specified **DirHandle** is invalid.

Remarks

When **DosFindClose** is issued, a subsequent **DosFindNext** for the closed **DirHandle** will fail unless an intervening **DosFindFirst** has been issued specifying **DirHandle**.

DosFindFirst - Find First Matching File

Purpose

Finds the first filename that matches the specified file specification.

Format

Calling Sequence:

```

EXTRN DosFindFirst:FAR

PUSH@ ASCIIZ FileName      ; File path name
PUSH@ WORD DirHandle       ; Directory search handle
PUSH WORD Attribute        ; Search attribute
PUSH@ OTHER ResultBuf     ; Result buffer
PUSH WORD ResultBufLen    ; Result buffer length
PUSH@ WORD SearchCount    ; # of entries to find
PUSH DWORD 0              ; Reserved (must be zero)
CALL DosFindFirst

EXTRN DosFindFirst2:FAR

PUSH@ ASCIIZ FileName      ; File path name
PUSH@ WORD DirHandle       ; Directory search handle
PUSH WORD Attribute        ; Search attribute
PUSH@ OTHER ResultBuf     ; Result buffer
PUSH WORD ResultBufLen    ; Result buffer length
PUSH@ WORD SearchCount    ; # of entries to find
PUSH WORD FileInfoLevel   ; File data required
PUSH DWORD 0              ; Reserved (must be zero)
CALL DosFindFirst2

```

Where

FileName is the ASCII path name of the file to be found.

DirHandle is the directory handle associated by the DOS with this specific request. A **DirHandle** value of 0x0001 is defined to be always available. A **DirHandle** value of 0xFFFF indicates to allocate a handle to the user. The handle is returned by overwriting the 0xFFFF. Reuse of this **DirHandle** in another **DosFindFirst** closes the association with the previously related **DosFindFirst** and opens a new association with the current **DosFindFirst**.

FileInfoLevel is the level of file information required. DOSFINDFIRST (and DOSFINDNEXT on handles returned by DOSFINDFIRST) always returns level 0x0001 information.

Level 'n' file information is returned in the format described in **ResultBuf**.

- If **FileInfoLevel** value is 0x0001, then if **Attribute** is set for hidden, system files, or directory files, it is considered as an **inclusive** search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, **Attribute** may be set to hidden + system + directory (all 3 bits on).

Attribute is the attribute used in searching for the file. **Attribute** cannot specify the volume label. Volume labels may be queried using **DosQFInfo**.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent **DosClose** or **DosBufReset**.

For level 0x0001, directory information (find record) is returned in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
    unsigned char   cbName;
    unsigned char   szName[];
};
```

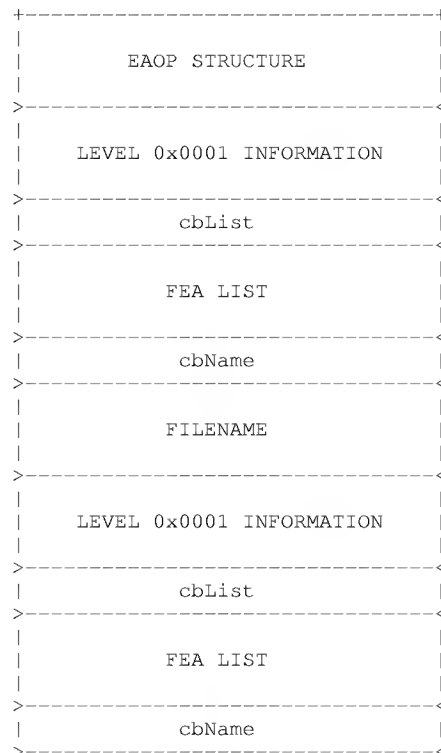
For level 0x0002, the cbList field is added to the level 0x0001 structure, and returned in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
    unsigned short  cbList;
    unsigned char   cbName;
    unsigned char   szName[];
};
```

The cbList field can be used to calculate the size of the buffer necessary for level 0x0003.

For level 0x0003, **ResultBuf** is an EAOP structure on input. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList and offError are ignored.

On output, **ResultBuf** will contain the EAOP structure (unchanged) followed by a packed set of records that consist of the attributes record described in level 0x0001 followed by an FEAList structure followed by a byte length of the name of the object followed by the asciiz name of the object matched by the input pattern.



- **ERROR_INVALID_EA_NAME** - there is an illegal character in an EA name. The **offError** field points to the offending GEA.
- **ERROR_EA_LIST_INCONSISTENT** - the **cbList** does not match the sum of the lengths of the GEA structures.

Remarks

DosFindNext uses the directory handle to repeat the related **DosFindFirst**.

The filename in **FileName** can contain wildcard characters.

DosFindNext - Find Next Matching File

Purpose

Finds the next directory entry matching the name that is specified on the previous **DosFindFirst** or **DosFindNext** function call.

Format

Calling Sequence:

EXTRN DosFindNext:FAR

```
PUSH WORD   DirHandle      ; Directory handle
PUSH@ OTHER ResultBuf      ; Result buffer
PUSH WORD   ResultBufLen   ; Result buffer length
PUSH@ WORD  SearchCount    ; # of entries to find
CALL  DosFindNext
```

Where

DirHandle is the handle (previously returned by DOS) associated with a previous **DosFindFirst** or **DosFindNext** function call.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent **DosClose** or **DosBufReset**.

For the continuation of an infolevel 3 search, this buffer should contain input in the same format as a **DosFindFirst2** infolevel 3 search.

For the continuation of an infolevel 4 search, this buffer should contain the **entryNum** in the same position as it is returned in by the **DosFindFirst2** infolevel 4 search. This infolevel is not for use by applications.

ResultBufLen is the length of **ResultBuf**.

SearchCount is the number of matching entries requested in **ResultBuf**. The file system uses this field to store the number of entries actually returned.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_PARAMETER** - a search count of zero is specified.
- **ERROR_INVALID_HANDLE** - the specified search handle is not in use.
- **ERROR_BUFFER_OVERFLOW** - the specified buffer is too small to support a single entry.
- **ERROR_NO_MORE_FILES** - there are no more files matching files.
- Device-driver/device-manager [errors listed](#).

Remarks

If no more matching files are found, an error code is returned.

DosFindNotifyClose - Close Find-Notify Handle

Purpose

Closes the association between a 'Find-Notify' handle and a **DosFindNotifyFirst** or **DosFindNotifyNext** function.

Format

Calling Sequence:

```
EXTRN DosFindNotifyClose:FAR

PUSH WORD DirHandle ; Find-notify handle
CALL DosFindNotifyClose
```

Where

DirHandle is the handle previously associated by the system with a **DosFindNotifyFirst** or **DosFindNotifyNext** function.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the specified handle is not in use.

Remarks

When **DosFindNotifyClose** is issued, a subsequent **DosFindNotifyNext** for the closed **DirHandle** will fail unless an intervening **DosFindNotifyFirst** has been issued specifying **DirHandle**.

DosFindNotifyFirst - Start monitoring directory for changes

Purpose

Start monitoring a directory for changes.

Format

Calling Sequence:

```
EXTRN DosFindNotifyFirst:FAR

PUSH@ ASCIIZ PathSpec      ; Path spec
PUSH@ WORD DirHandle       ; Path handle
PUSH WORD Attribute        ; Search attribute
PUSH@ OTHER ResultBuf      ; Result buffer
PUSH WORD ResultBufLen     ; Result buffer length
PUSH@ WORD ChangeCount     ; # of changes required
PUSH WORD FileInfoLevel    ; File data required
```

```

PUSH  DWORD  Timeout      ; Duration of call
PUSH  DWORD  0            ; Reserved (must be zero)
CALL  DosFindNotifyFirst

```

Where

PathSpec is the path name to be monitored for directory changes.

DirHandle is the handle returned by OS/2 which is associated with this request.

FileInfoLevel is the level of file information required. A value of 0x0001 returns no information to **ResultBuf**. Values other than 0x0001 are not supported in this release of OS/2.

Attribute is the file attribute used in qualifying the files and/or directories to be monitored. **Attribute** cannot specify the volume label. Volume labels may be queried using **DosQFInfo**.

ResultBuf is where the filesystem returns the results of the **PathSpec** monitoring.

ResultBufLen is the length of **ResultBuf**.

ChangeCount is the number of changes required to directories or files that match the **PathSpec** target and **Attribute**. The file system uses this field to return the number of changes that actually occurred.

Timeout is the maximum duration of **DosFindNotifyFirst** before returning to the function caller.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the specified handle is not in use.
- ERROR_NO_MORE_SEARCH_HANDLES - too many search handles are currently in use by the process.
- ERROR_PATH_NOT_FOUND - the drive letter is invalid, there are too many ..., there are wildcards present before the last component, or a component of the directory path is not present.
- ERROR_INVALID_PARAMETER - the search attribute is invalid or a change count of zero is specified.
- ERROR_BUFFER_OVERFLOW - the specified buffer is too small to support a single entry.
- ERROR_FINDNOTIFY_TIMEOUT - the timeout expired without the requested number of changes occurring.
- Device-driver/device-manager [errors listed](#).

Remarks

The find notify API allow a desktop to be informed of changes in a directory in order to provide an up-to-date view of the directory. Changes consist of changes to names within a directory (file create, file delete, file/directory rename, directory create, directory removal) plus operations that change a file attribute. Those operations are SetFileMode, SetFileInfo, and SetPathInfo for both standard attributes and extended attributes.

Changes to date and time and size will NOT be reported for each read or write, but will instead be reported when a file is closed or committed.

DosFindNotifyFirst will block until either the timeout specified has elapsed or until the specified number of changes has been found.

DosFindNotifyNext - Return directory change information.

Purpose

Return directory change information.

Format

Calling Sequence:

EXTRN DosFindNotifyNext:FAR

```
PUSH WORD DirHandle ; Directory handle
PUSH@ OTHER ResultBuf ; Result buffer
PUSH WORD ResultBufLen ; Result buffer length
PUSH@ WORD ChangeCount ; # of changes required
PUSH DWORD TimeOut ; Duration of call
CALL DosFindNotifyNext
```

Where

DirHandle is the handle (previously returned by DOS) associated with a previous **DosFindNotifyFirst** or **DosFindNotifyNext** function call.

ResultBuf is where the FSD returns the results of the qualified directory search. The information returned is guaranteed to be at least as accurate as the most recent **DosClose** or **DosBufReset**.

ResultBufLen is the length of **ResultBuf**.

ChangeCount is the number of changes required to directories or files that match the **PathSpec** target and **Attribute** specified during a related, previous **DosFindNotifyFirst**. The file system uses this field to return the number of changes that actually occurred since the present **DosFindNotifyNext** was issued.

TimeOut is the maximum duration of **DosFindNotifyFirst** before returning to the function caller.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the specified handle is not in use.
- ERROR_INVALID_PARAMETER - a change count of zero is specified.
- ERROR_BUFFER_OVERFLOW - the specified buffer is too small to support a single entry.
- ERROR_FINDNOTIFY_TIMEOUT - the timeout expired without the requested number of changes occurring.
- Device-driver/device-manager [errors listed](#).

Remarks

DosFindNotifyNext will block until either the timeout specified has elapsed or until the specified number of changes has been found.

DosFsAttach - Creates and destroys FSD connections.

Purpose

Attaches or detaches drive to/from a remote FSD, or a pseudo-character device name to/from a local or remote FSD

Format

Calling Sequence:

EXTRN DosFsAttach:FAR

```
PUSH@ ASCIIZ DeviceName ; Device name or 'd:'
PUSH@ ASCIIZ FSDName ; FSD name
```

```

PUSH@ OTHER  DataBuffer      ; Attach argument data
PUSH  WORD   DataBufferLen   ; Buffer length
PUSH  WORD   OpFlag          ; Attach or detach
PUSH  DWORD  0                 ; Reserved (must be zero)
CALL  DosFsAttach

```

Where

DeviceName points to either the drive letter followed by a colon, or points to a pseudo-character device name. If **DeviceName** is a drive, it is an ASCIIZ string having the form of drive letter followed by a colon. If an attach is successful, all requests to that drive are routed to the specified FSD. If a detach is successful, the drive will disappear from the system's name space.

If **DeviceName** is a pseudo-character device name (i.e., single file device), its format is that of an ASCIIZ string in the format of an OS/2 filename in a subdirectory called **\DEV**. All requests to that name are routed to the specified FSD after a successful attach. A successful detach removes the name from the system's name space.

FSDName is the ASCIIZ name of the **remote** FSD to attach to or detach from **DeviceName**.

DataBuffer points to the user-supplied FSD argument data area. The meaning of the data is specific to the FSD. The **DataBuffer** should contain contiguous asciiz strings, with the first word of the buffer containing the number of asciiz strings.

DataBufferLen is the byte length of the data buffer.

OpFlag defines the type of operation to be performed.

- Attach = 0
- Detach = 1

Returns:

AX = 0

ReturnCode = 0 if no error

ReturnCode = Error code if error

- ERROR_INVALID_DRIVE - the drive specified in **DeviceName** is illegal.
- ERROR_INVALID_PATH - the pseudodevice specified in **DeviceName** is illegal.
- ERROR_INVALID_FSD_NAME - the FSD name specified is not found.
- ERROR_INVALID_LEVEL - the value of **OpFlag** is invalid.

Remarks

The redirection of drive letters representing local drives is not supported.

FSDs that wish to establish open connections that are not attached to a name in the system's name space, for such purposes as optimizing UNC connections or establishing access rights, must use an **DosFsCtl DosFsAttach** only creates attachments to drives or devices in the system's name space.

See description of pseudo-character devices.

DosFsCtl - File System Control

Purpose

Allows an extended standard interface between an application and an FSD.

Format

Calling Sequence:

```

EXTRN DosFsCtl:FAR

PUSH@ OTHER DataArea      ; Data area
PUSH WORD DataAreaLength ; Data area length
PUSH@ OTHER ParmList      ; Parameter list
PUSH WORD ParmListLength  ; Parameter list length
PUSH WORD FunctionCode    ; Function code
PUSH@ ASCIIZ RouteName    ; Path or FSD name
PUSH WORD FileHandle      ; File handle
PUSH WORD RouteMethod     ; Method for routing.
PUSH DWORD 0              ; Reserved (must be zero)
CALL DosFsCtl

```

Where

DataArea is a data area.

DataAreaLength is the length in bytes of the buffer specified by **DataArea**.

ParmList is a command specific parameter list.

ParmListLength is the length of **ParmList**.

FunctionCode is the filesystem-specific function code. Function codes between 0x0000 and 0x7FFF are reserved for use by OS/2.

RouteName is the ASCIIZ name of the FSD, or the pathname of a file or directory the operation should apply to.

FileHandle is the file or device specific handle.

RouteMethod selects how the request will be routed.

- **RouteMethod = 1:** **FileHandle** **RouteName** must be a NUL pointer (0L). The FSD associated with the handle will receive the request.
- **RouteMethod = 2:** **RouteMethod** refers to a pathname, which directs routing. **FileHandle** must be -1. The FSD associated with the drive the pathname refers to at the time of the request will receive the request. The pathname need not refer to a file or directory which actually exists, only to a drive which does. A relative pathname may be used, it will be processed like any other pathname.
- **RouteMethod = 3:** **RouteMethod** refers to an FSD name, which directs routing. **FileHandle** must be -1. The named FSD will receive the request.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_FUNCTION** - the specified function is illegal for the particular category and handle.
- **ERROR_INVALID_CATEGORY** - the specified category is illegal for the particular function and handle.
- **ERROR_INVALID_PARAMETER** - the filehandle != -1 when the routemethod == 2 or the routename != 0 when the routemethod == 1.
- **ERROR_INVALID_HANDLE** - the specified file handle is not in use or is attached to a physical device.
- **ERROR_INVALID_FSD_NAME** - the specified FSDName is not responsible for managing the specified file handle.
- **ERROR_INVALID_LEVEL** - invalid route method.
- **ERROR_INTERRUPT** - the current thread received a signal.

Remarks

For func == 1, the error code is passed to the FSD in the first word of the parameter area. On return, the first word of the data area contains the length of the ascii string containing an explanation of the error code. The data area contains the ascii string beginning at the second word. All FSDs are required to support func == 1.

DosMkDir - Make Subdirectory

Purpose

Creates the specified directory.

Format

Calling Sequence:

EXTRN DosMkDir:FAR

```
PUSH@ ASCIIZ DirName    ; New directory name
PUSH  DWORD 0           ; Reserved (must be zero)
CALL  DosMkDir
```

EXTRN DosMkDir2:FAR

```
PUSH@ ASCIIZ DirName    ; New directory name
PUSH@ OTHER EABuf       ; Extended attribute buffer
PUSH  DWORD 0           ; Reserved (must be zero)
CALL  DosMkDir2
```

Where

DirName is the ASCIIZ directory path name.

EABuf contains, on input, an EAOP structure. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

If EABuf is 0x00000000, then no extended attributes are defined for the directory.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_PATH_NOT_FOUND** - the drive letter is invalid, there are too many ..., there are wildcards present, or a component of the directory path is not present.
- **ERROR_FILENAME_EXCED_RANGE** - the path specified is unacceptable to the FSD managing the volume.
- **ERROR_ACCESS_DENIED** - the directory already exists or a file with this name exists.
- Device-driver/device-manager [errors listed](#).
- **ERROR_INVALID_EA_NAME** - there is an illegal character in an EA name. The offError field points to the offending FEA.
- **ERROR_EA_LIST_INCONSISTENT** - the cbList does not match the sum of the lengths of the FEA structures.
- **ERROR_EA_VALUE_UNSUPPORTED** - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.

Remarks

If any member of the directory path does not exist, then the directory path is not created. On return, a new directory is created at the end of the specified path.

DosMove - Move a File or a Subdirectory

Purpose

Moves the specified file or subdirectory.

Format

Calling Sequence:

EXTRN DosMove:FAR

```
PUSH@ ASCIIZ OldPathName ; Old path name
PUSH@ ASCIIZ NewPathName ; New path name
PUSH  DWORD 0           ; Reserved (must be zero)
CALL  DosMove
```

Where

OldPathName is the old ASCIIZ path name of the file or subdirectory to be moved.

NewPathName is the new ASCIIZ path name of the file or subdirectory to be moved.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_PATH_NOT_FOUND - the drive letter is invalid, there are too many .., there are wildcards present, or a component of the directory path is not present.
- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR_SHARING_VIOLATION - the OldPathName is currently open.
- ERROR_FILE_NOT_FOUND - the OldPathName is not present
- ERROR_NOT_SAME_DEVICE - the OldPathName and NewPathName are on different physical devices.
- ERROR_ACCESS_DENIED - NewPathName already exists, OldPathName is a character device, or OldPathName is a pseudocharacter device.
- ERROR_CIRCULARITY_REQUESTED - the OldPathName is an ancestor of the NewPathName.
- ERROR_DIRECTORY_IN_CDS - the OldPathName is the current directory of a process.
- ERROR_SHARING_BUFFER_EXCEEDED - there is not enough memory to hold sharing information.
- Device-driver/device-manager [errors listed](#).

Remarks

The directory paths need not be the same, allowing a file or subdirectory to be moved to another directory and renamed in the process.

Wildcard characters are not allowed in the source or destination name.

File systems will reject requests that move a parent directory to one of its subdirectories in order to avoid a loop in the directory tree. A subdirectory cannot be both a descendent and an ancestor of the same directory.

An attempt to move the current directory or any of its ancestors for the current or any other process will fail and cause the operation to terminate.

An attempt to move the current directory for any process will fail and cause the operation to terminate.

Read-only files in the target path are not replaced.

DosMove will move the source's attributes (date of creation, time of creation, ...) to the target.

DosNewSize - Change File's Size

Purpose

Changes a file's logical (EOD) size.

Format

Calling Sequence:

```
EXTRN DosNewSize:FAR

PUSH WORD   FileHandle ; File handle
PUSH DWORD  FileSize   ; File's new size
CALL  DosNewSize
```

Where

FileHandle is the handle of the file whose size is being changed.

FileSize is the file's new size in bytes.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_DISK_FULL - there is not enough room to grow the file.
- ERROR_INVALID_PARAMETER - the size is negative.
- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR_ACCESS_DENIED - this handle is opened read-only.
- ERROR_LOCK_VIOLATION - the region of file growth overlaps a locked region.
- Device-driver/device-manager [errors listed](#).

Remarks

The file handle must be opened for read/write or write-only access.

The value of new bytes in the extended file is undefined.

The file system will make a reasonable attempt to allocate the new size in a contiguous (or nearly contiguous) space on the media.

DosOpen - Open a File

Purpose

Creates the specified file (if necessary), and opens it.

Format

Calling Sequence:

EXTRN DosOpen:FAR

```
PUSH@ ASCIIZ FileName      ; File path name
PUSH@ WORD   FileHandle     ; New file's handle
PUSH@ WORD   ActionTaken    ; Action taken
PUSH  DWORD  FileSize       ; File primary allocation
PUSH  WORD   FileAttribute   ; File Attribute
PUSH  WORD   OpenFlag        ; Open function type
PUSH  WORD   OpenMode        ; Open mode of the file
PUSH  DWORD  0               ; Reserved (must be zero)
CALL  DosOpen
```

EXTRN DosOpen2:FAR

```
PUSH@ ASCIIZ FileName      ; File path name
PUSH@ WORD   FileHandle     ; New file's handle
PUSH@ WORD   ActionTaken    ; Action taken
PUSH  DWORD  FileSize       ; File primary allocation
PUSH  WORD   FileAttribute   ; File Attribute
PUSH  WORD   OpenFlag        ; Open function type
PUSH  WORD   OpenMode        ; Open mode of the file
PUSH@ OTHER  EABuf          ; EA buffer
PUSH  DWORD  0               ; Reserved (must be zero)
CALL  DosOpen2
```

Where

FileName is the ASCII path name of the file or device name to be opened.

FileHandle is where the system returns the file handle.

ActionTaken is where the system returns a description of the action taken as a result of the **DosOpen** function call.

- 0x0001 file existed
- 0x0002 file was created
- 0x0003 file was replaced

FileSize is the new file's logical size (EOD) in bytes.

FileAttribute is the file attribute. Refer to **DosQFileMode** or **DosSetFileMode** for a description of **FileAttribute**.

OpenFlag specifies the action to be taken depending on whether or not the file exists.

- **OpenFlag** specification:
Low Order Byte

----	xxxx	action taken if file exists	----	0000	fail	----	0001	open file	----	0010	replace
file											
xxxx	----	action taken if file			doesn't exist	0000	----	fail	0001	----	create file

OpenMode is the open mode and consists of the following bit fields. They are the:

- Inheritance flag
- Write-through flag
- Fail-errors flag
- Sharing mode field
- Access field
- Reserved bit fields

The bit field mapping is shown as follows:

1 1 1 1 1 Open Mode 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 bits D W F R R L L L I S S S R A A A

D

Direct Open

The file is opened as follows:

If D = 0; FileName represents a file to be opened in the normal way. If D = 1; FileName is "<Drive>:" and represents a **mounted** disk or diskette volume to be opened for direct access.

W

File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run through the DOS buffer cache. If W = 1; Writes to the file may go through the DOS buffer cache but sectors will be written (actual file I/O completed) before a synchronous write call returns. This state of the file defines it as a synchronous file.

This bit is not inherited by child processes.

I

Inheritance Flag

If I = 0; File handle is inherited by a spawned process resulting from a **DosExecPgm** call. If I = 1; File handle is private to the current process.

This bit is not inherited by child processes.

F

Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system critical error handler. If F = 1; reported directly to the caller via return code.

This bit is not inherited by child processes. Media I/O errors generated through an IoCtl Category 8 function, always get reported directly to the caller via return code. The **Fail-Errors** functionality applies only to non-IoCtl handle-based type file I/O calls.

R

Reserved and must-be-zero field

L

Locality of reference

The locality-of-reference is advisory information about how the application will access the file.

If L == 00; no locality known If L == 01; mainly sequential access If L == 10; mainly random access If L == 11; random with some locality

S

Sharing Mode

The file sharing mode field defines what operations other processes may perform on the file.

If S = 001; Deny Read/Write access If S = 010; Deny Write access If S = 011; Deny Read access If S = 100; Deny Neither Read or Write access (Deny None)

Any other value is invalid.

A

Access Mode

The file access is assigned as follows:

If A = 000; Read-only access If A = 001; Write-only access If A = 010; Read/Write access

Any other value is invalid.

Any other combinations are invalid.

When opening a file, it is important to inform OS/2 what operations other processes may perform on this file (sharing mode). If it is permissible for other processes to continue to read this file while your process is operating on the file, you should specify Deny Write, which inhibits writing by other processes, but allows reading by them.

Similarly, it is important to specify what operations your process will perform (access mode). The Read/Write access mode causes the open request to fail if another process has the file opened with any sharing mode other than deny none. If however, all you intended to do is read from the file, your open will not succeed unless all other processes

have specified deny none or deny write (therefore increasing access to the file). File sharing requires cooperation of both sharing processes. This cooperation is communicated through the sharing and access mode.

EABuf contains, on input, an EAOP structure. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

If EABuf is 0x00000000, then no extended attributes are defined for the file.

If **Extended Attributes** are not to be defined or modified, then the pointer, **EABuf**, must be set to null.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_PARAMETER** - the OpenFlag value is invalid, the OpenMode value is invalid, the OpenFlag/OpenMode values are incompatible, or the size specified in a replace/create operation is negative.
- **ERROR_INVALID_ACCESS** - the access mode field is invalid.
- **ERROR_ACCESS_DENIED** - the attributes specified are invalid, the file is read only and OpenMode requests a write access, or the FileName points to a directory.
- **ERROR_OPEN_FAILED** - the combination of OpenFlag and state of the file results in a failed operation.
- **ERROR_DISK_FULL** - there is not enough room on the media to create the file with the specified size.
- **ERROR_TOO_MANY_OPEN_FILES** - there are too many files open by either the system or the process.
- **ERROR_FILE_NOT_FOUND** - the last component of the file name is not present on the media.
- **ERROR_PATH_NOT_FOUND** - a component of the directory path is not present.
- **ERROR_SHARING_BUFFER_EXCEEDED** - there is not enough memory to hold sharing information.
- **ERROR_SHARING_VIOLATION** - the FileName is currently open.
- **ERROR_FILENAME_EXCED_RANGE** - the path specified is unacceptable to the FSD managing the volume.
- Device-driver/device-manager errors listed.
- **ERROR_INVALID_EA_NAME** - there is an illegal character in an EA name. The offError field points to the offending FEA.
- **ERROR_EA_LIST_INCONSISTENT** - the cbList does not match the sum of the lengths of the FEA structures.
- **ERROR_EA_VALUE_UNSUPPORTED** - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.

Remarks

The read/write pointer is set at the first byte of the file. The read/write pointer can be changed through function **DosChgFilePtr**.

The file's date and time can be obtained through function **DosQFileInfo**.

The file's date and time can be set through function **DosSetFileInfo**, and its attribute can be obtained through function **DosQFileMode**.

The **FileSize** parameter affects the size of the file only when it is created or replaced. If an existing file is simply opened, **FileSize** is ignored.

The **Direct Open** bit parameter is the "Direct I/O flag". It provides an access mechanism to an entire disk or diskette volume independent of the file system. This mode of opening the volume currently mounted on the drive, is used to return a handle to the caller which represents the logical volume as a single file. In order to block other processes from

accessing the logical volume, the caller must also issue a **DosDevIoCtl** Category 8 sub-function 0, which requires the file handle for the logical volume returned by **DosOpen**.

The file handle state bits can be set by the **DosOpen** and **DosSetFHState** function calls. An application may query the file handle state bits as well as the rest of the **Open Mode** field, by using the **DosQFHandState** function call.

The returned file handle must be used for subsequent input and output to the file.

The value of new bytes in the extended file is undefined.

The file system must make a reasonable attempt to allocate the new size in a contiguous (or nearly contiguous) area on the media. Extended attributes that indicate required contiguity may reject the call if unable to allocate contiguous space.

The extended attributes will be set for creation of a new file, replacement of an existing file, and truncation of an existing file. No attributes are set for a normal open.

FileAttribute cannot be set to Volume Label. Volume labels cannot be opened.

Notes

Notes:

1. A multitasking system must be able to use files and their existence as semaphores. This function call may be used as a test-and-set semaphore when used to create a new file.
2. When a file is closed, any sharing restrictions placed on it by the open are removed.
3. The file read-only attribute can be set by using the **DosSetFileMode** function call or the OS/2 ATTRIB command.
4. If the file is inherited by a spawned process, all sharing and access restrictions are also inherited.
5. If an open file handle is duplicated by function call **DosDupHandle**, all sharing and access restrictions are also duplicated.

Sharing Modes

Deny Read/Write Mode

If a file is successfully opened in Deny Read/Write mode, access to the file is exclusive. A file currently open in this mode cannot be opened again in any sharing mode by any process (including the current process) until the file is closed.

Deny Write Mode

A file successfully opened in Deny Write sharing mode, prevents any other write access opens to the file (A = 001 or 010) until the file is closed. An attempt to open a file in Deny Write mode is unsuccessful if the file is currently open with a write access.

Deny Read Mode

A file successfully opened in Deny Read sharing mode, prevents any other read sharing access opens to the file (A = 000 or 010) until the file is closed. An attempt to open a file in Deny Read sharing mode is unsuccessful if the file is currently open with a read access.

Deny None Mode

A file successfully opened in Deny None mode, places no restrictions on the read/write accessibility of the file.

Named Pipe Considerations Opens the client end of a pipe by name and returns a handle. The pipe must be in listen state for the open to succeed; otherwise the open fails with PIPE BUSY (this happens, for example, if all instances of the pipe are already open, if the pipe is closed but not yet disconnected by the serving end, or if no **DosConnectNmPipe** has been issued against the pipe since it was last disconnected). Once a given instance has been opened by a client that same instance cannot be opened by another client at the same time (i.e., pipes can only be two-ended at present); the opening process can of course dup the open handle as many times as desired. The access and sharing modes specified on the open must be consistent with those specified on the **DosMakeNmPipe**. Pipes are always opened with the pipe-specific states set to B = 0 (blocking reads/writes), RR = 00 (read as byte stream). The client can change these modes via **DosSetPHandState** if desired.

DosQCurDir - Query Current Directory

Purpose

Gets the full path name of the current directory for the requesting process for the specified drive.

Format

Calling Sequence:

```
EXTRN DosQCurDir:FAR

PUSH WORD DriveNumber ; Drive number
PUSH@ OTHER DirPath    ; Directory path buffer
PUSH@ WORD DirPathLen   ; Directory path buffer length
CALL  DosQCurDir
```

Where

DriveNumber is the drive number (0=default, 1=A, ...).

DirPath is where the system returns the full directory path name.

DirPathLen is the length of the **DirPath** buffer.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_DRIVE - the specified drive is invalid.
- ERROR_BUFFER_OVERFLOW - the current directory is too long to fit into the specified directory.
- Device-driver/device-manager [errors listed](#).

Remarks

The drive letter is not part of the returned string. The string does not begin with a backslash and is terminated by a byte containing 0x00.

The system returns the length of the returned **DirPath** string (not including the 0x00 terminating null byte) in **DirPathLen**.

In the case where the **DirPath** buffer is of insufficient length to hold the current directory path string, the system returns the required length (in bytes) for **DirPath** in **DirPathLen**.

DosQCurDisk - Query Current Disk

Purpose

Determine the current default drive for the requesting process.

Format

Calling Sequence:

```
EXTRN DosQCurDisk:FAR

PUSH@ WORD DriveNumber ; Default drive number
PUSH@ DWORD LogicalDriveMap ; Drive-map area
CALL  DosQCurDisk
```

Where

DriveNumber is where the system returns the number of the default drive (1=A, 2=B, ...)

LogicalDriveMap is a bit map (stored in the low-order portion of the 32-bit double word area) in which the system returns the mapping of the logical drives. Logical Drives A-Z have a one-to-one mapping with the bit positions 0-25 of the map.

- If bit value = 0, the logical drive does not exist
- If bit value = 1, the logical drive exists

Returns:

No error returns are defined for this API.

DosQFHandState - Query File Handle State

Purpose

Query the state of the specified file.

Format

Calling Sequence:

EXTRN DosQFHandState:FAR

```
PUSH WORD FileHandle ; File handle
PUSH@ WORD FileHandleState ; File handle state
CALL DosQFHandState
```

Where

FileHandle is the handle of the file to be queried.

FileHandleState is the file handle state and consists of the following bit fields. They are the:

- Inheritance flag
- Write-through flag
- Fail-errors flag
- Sharing mode field
- Access field
- Reserved bit fields

The bit field mapping is shown as follows:

```
.br
Open Mode bits  5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
                D W F R R R R R I S S S R A A A
```

D

Direct Open

The file is opened as follows:

```
If D = 0; Pathname represents a file
          opened in the normal way.
If D = 1; Pathname is "<Drive>:" and
          represents a mounted disk or diskette
          volume to opened for direct access.
```

L

Inheritance Flag

If I = 0; File handle is inherited by a spawned process resulting from a **DosExecPgm** call.
If I = 1; File handle is private to the current process.

This bit is not inherited by child processes.

W

File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run through the DOS buffer cache.
If W = 1; Writes to the file may go through the DOS buffer cache but sectors will be written (actual file I/O completed) before a synchronous write call returns. This state of the file defines it as a synchronous file.

This bit is not inherited by child processes.

E

Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system critical error handler.
If F = 1; reported directly to the caller via return code.

This bit is not inherited by child processes. Media I/O errors generated through an IoCtl Category 8 function, always get reported directly to the caller via return code. The **Fail-Errors** functionality applies only to non-IoCtl handle-based type file I/O calls.

R

These bits are reserved and should be set to the values returned by **DosQFHandState** in these positions.

S

Sharing Mode

The file sharing mode field defines what operations other processes may perform on the file.

If S = 001; Deny Read/Write access
If S = 010; Deny Write access
If S = 011; Deny Read access
If S = 100; Deny Neither Read or Write access (Deny None)

Any other value is invalid.

A

Access Mode

The file access is assigned as follows:

If A = 000; Read-only access
If A = 001; Write-only access
If A = 010; Read/Write access

Any other value is invalid.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the handle is not in use or is attached to a physical device.

Remarks

Named Pipe Considerations

As defined by OS/2. D = 0. Other bits as defined by DosMakeNmPipe (serving end), DosOpen (client end), or the last DosSetFHandState.

DosQFileInfo - Query a File's Information

Purpose

Returns information for a specific file.

Format

Calling Sequence:

```
EXTRN DosQFileInfo:FAR

PUSH WORD FileHandle      ; File handle
PUSH WORD FileInfoLevel   ; File data required
PUSH@ OTHER FileInfoBuf   ; File data buffer
PUSH WORD FileInfoBufSize ; File data buffer size
CALL DosQFileInfo
```

Where

FileHandle is the file handle.

FileInfoLevel is the level of file information required.

File information, where applicable, is at least as accurate as the most recent **DosClose**, **DosBufReset**, or **DosSetFileInfo**.

Level 0x0001 information is returned in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
};
```

Level 0x0002 simply adds cbList to level 0x0001, returning a structure in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
    unsigned short  cbList;
};
```

Level 0x0003 file information returns a subset of the EA information for the file. On input, FileInfoBuf is an EAOP structure above. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, FileInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR_BUFFER_OVERFLOW) cbList will still be valid, assuming there's at least enough space for it. Its value will be the size of the entire EA set for the file, even though only a subset of attributes was requested.

Level 0x0004 file information returns all EA info for the file. On input, FileInfoBuf is an EAOP structure above. fpGEAList contents are ignored. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, FileInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR_BUFFER_OVERFLOW) cbList will still be valid.

FileInfoBuf is the storage area where the system returns the requested level of file information.

FileInfoBufSize is the length of **FileInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_LEVEL - the specified FileInfoLevel is not valid.
- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR_DIRECT_ACCESS_HANDLE - the specified handle is opened with the direct open bit set.
- ERROR_BUFFER_OVERFLOW - the specified buffer is not long enough to return the desired information.
- Device-driver/device-manager [errors listed](#).
- ERROR_INVALID_EA_NAME - there is an illegal character in an EA name. The offError field points to the offending GEA.
- ERROR_EA_LIST_INCONSISTENT - the cbList does not match the sum of the lengths of the GEA structures.

Remarks

The FAT file system supports only the modification date and time. Zero will be returned for the creation and access dates and times.

Purpose

Query the mode (attribute) of the specified file.

Format

Calling Sequence:

```
EXTRN DosQFileMode:FAR

PUSH@ ASCIIZ FilePathName      ; File path name
PUSH@ WORD   CurrentAttribute   ; Data area
PUSH  DWORD  0                  ; Reserved (must be zero)
CALL  DosQFileMode
```

Where

FilePathName is the ASCIIZ file path name.

CurrentAttribute is where the system returns the file's current attribute.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_PATH_NOT_FOUND - a component of the directory path is not present.
- ERROR_FILE_NOT_FOUND - the last component of the file name is not present on the media.
- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- Device-driver/device-manager [errors listed](#).

Remarks

The 'Volume Label' type attribute is not returned by **DosQFileMode**, **DosQFsInfo** may be used for this purpose.

File attributes are defined as follows.

<u>0x0001</u>	read only file.
<u>0x0002</u>	hidden file.
<u>0x0004</u>	system file.
<u>0x0010</u>	subdirectory.
<u>0x0020</u>	file archive.

All other attributes are reserved.

DosQFsAttach - Query attached FSD information

Purpose

Query information about an attached remote file system or a local file system or about a character device or about pseudo-character device name attached to a local or remote FSD.

Format

Calling Sequence:

```
EXTRN DosQFsAttach:FAR

PUSH@ ASCIIZ DeviceName    ; Device name or 'd:'
PUSH  WORD  Ordinal        ; Ordinal of entry in name list
PUSH  WORD  FSAInfoLevel   ; Type of attached FSD data required
PUSH@ OTHER  DataBuffer     ; Returned data buffer
PUSH@ WORD  DataBufferLen   ; Buffer length
PUSH  DWORD 0              ; Reserved (must be zero)
CALL  DosQFsAttach
```

Where

DeviceName points to the drive letter followed by a colon, or points to a character or pseudo-character device name, or is ignored for some values of **FSAInfoLevel**. If **DeviceName** is a drive, it is an ASCIIZ string having the form of drive letter followed by a colon. If **DeviceName** is a character or pseudo-character device name its format is that of an ASCIIZ string in the format of a OS/2 filename in a subdirectory called **\DEV**.

Ordinal is an index into the list of character or pseudo-character devices, or the set of drives. **Ordinal** always starts at 1. The **Ordinal** position of an item in a list has no significance at all, **Ordinal** is used strictly to step through the list. The mapping from **Ordinal** to item is volatile, and may change from one call to **DosQFsAttach** to the next.

FSAInfoLevel is the level of information required, and determines which item the data in **DataBuffer** refers to.

Level 0x0001 returns data for the specific drive or device name referred to by **DeviceName**. The **Ordinal** field is ignored.

Level 0x0002 returns data for the entry in the list of character or pseudo-character devices selected by **Ordinal**. The **DeviceName** field is ignored.

Level 0x0003 returns data for the entry in the list of drives selected by **Ordinal**. The **DeviceName** field is ignored.

DataBuffer is the return information buffer, it is in the following format:

```
struct {
    unsigned short iType;
    unsigned short cbName;
    unsigned char  szName[];
    unsigned short cbFSDName;
    unsigned char  szFSDName[];
    unsigned short cbFSADData;
    unsigned char  rgFSADData[];
};
```

iType

type of item

- 1 = Resident character device
- 2 = Pseudo-character device
- 3 = Local drive
- 4 = Remote drive attached to FSD

cbName

Length of item name, not counting null.

szName

Item name, ASCIIZ string.

cbFSDName

Length of FSD name, not counting null.

szFSDName

Name of FSD item is attached to, ASCIIZ string.

cbFSADData

Length of FSD Attach data returned by FSD.

rgFSADData

FSD Attach data returned by FSD.

szFSDName is the FSD name exported by the FSD, which is not necessarily the same as the FSD name in the boot sector.

For local character devices (iType = 1), cbFSDName = 0 and szFSDName will contain only a terminating NULL byte, and cbFSADData = 0.

For local drives (iType = 3), szFSDName will contain the name of the FSD attached to the drive at the time of the call. This information changes dynamically. If the drive is attached to the kernel's resident file system, szFSDName will contain "**FAT**".

DataBufferLen is the byte length of the return buffer. Upon return, it is the length of the data returned in **DataBuffer** by the FSD.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_DRIVE - the drive specified is invalid
- ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.
- ERROR_NO_MORE_ITEMS - the **Ordinal** specified refers to an item not in the list.
- ERROR_INVALID_LEVEL - invalid info level

Remarks

Information about all block devices and all character and pseudo-character devices is returned by this call.

The information returned by this call is highly volatile. Calling programs should be aware the the returned information may have already changed by the time it's returned to them.

DosQFInfo - Query File System Information

Purpose

Query information from a file system.

Format

Calling Sequence:

EXTRN DosQFInfo:FAR

```
PUSH WORD DriveNumber    ; Drive number
PUSH WORD FSInfoLevel     ; File system data level required
PUSH@ OTHER FSInfoBuf     ; File system info buffer
PUSH WORD FSInfoBufSize   ; File system info buffer size
CALL DosQFInfo
```

Where

DriveNumber is the logical drive number (0=default, 1=A,2=B,3=C, ...) and represents the FSD for the media currently in that drive or the FSD that is currently attached with that drive.

When a logical drive is specified, the media in the drive is examined (local drive only) and the request is passed to the FSD responsible for managing that media or to the FSD that is attached to the drive (remote case).

FSInfoLevel is the level of file information required.

Level 0x0001 information is returned in the following format:

```
struct {
```

```

        unsigned long    reserved;
        unsigned long    csecPerAlloc;
        unsigned long    callocTotal;
        unsigned long    callocFree;
        unsigned short   cbPerSec;
};

```

Level 0x0002 information is returned in the following format:

```

struct {
    unsigned long    ulVSN;
    unsigned char    cbVolLabel;
    unsigned char    szVolLabel[];
};

```

ulVSN
volume serial number

cbVolLabel
length of volume label

szVolLabel
asciiz volume label

FSInfoBuf is the storage area where the system returns the requested level of file system information.

FSInfoBufSize is the length of **FSInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_DRIVE - the drive specified is invalid
- ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.
- ERROR_INVALID_LEVEL - the specified FSInfoLevel is not supported.
- Device-driver/device-manager [errors listed](#).

Remarks

Trailing blanks supplied at volume label definition time are not considered to be part of the label and are therefore not returned as valid label data.

Volume Serial Number is a unique 32-bit number used by OS/2 to positively identify its disk/diskette volumes. The hard error daemon will prompt the user for an unmounted removable volume by displaying both the **Volume Serial Number** (as an 8 digit hexadecimal number) and the **Volume Label**.

If there is no volume serial number on the disk/diskette, the volume serial number information will be returned as binary zeros.

If there is no volume label on the disk/diskette, the volume label information will be returned as blank spaces.

If there is no volume serial number and/or volume label for disk/diskette volumes formatted by DOS 3.X, this information is not displayed by the Hard Error Handler.

DosQHandType - Query a Handle Type

Purpose

Determines whether a handle references a file or a device.

Format

Calling Sequence:

```
EXTRN DosQHandType:FAR

PUSH WORD FileHandle      ; File handle
PUSH@ WORD HandType       ; Handle type response
PUSH@ WORD FlagWord       ; Device descriptor word
CALL DosQHandType
```

Where

FileHandle is the file handle.

HandType is where the system returns the value indicating the handle type. **HandType** is composed of two bytes:

HandleClass

Describes the handle class. It may take on the following values in the low byte of **HandType**:

- 0, handle is for a disk file.
- 1, handle is for a character device.
- 2, handle is for a pipe.

Values greater than 2 are reserved.

HandleBits

Provides further information about the handle in the high byte of **HandType**. This byte is broken into eight bits, whose meaning depends upon the value of **HandleClass**:

	HandleBits								HandleClass	
	5	4	3	2	1	0	9	8	7	----- 0
Disk file	N	u	u	u	u	u	u	u		0
Char device	N	u	u	u	u	u	u	u		1
Pipe	N	u	u	u	u	u	u	u		2

N is the **NETWORK** bit. If set, it means that the handle refers to a remote file, device, or pipe. Otherwise, the handle refers to a local file, device, or pipe.

u means that the bit is undefined and reserved. A program may not depend upon the values of these bits, as they may change in future versions of OS/2.

FlagWord is where the system returns the device's driver attribute word if **HandType** indicates a local character device.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.

Remarks

This function allows some programs which may be interactive or file-oriented to determine the source of their input. For example, CMD.EXE suppresses writing prompts if the input is from a disk file.

DosQPathInfo - Query a file or a subdirectory for information

Purpose

Returns information for a specific file or subdirectory.

Format

Calling Sequence:

```
EXTRN DosQPathInfo:FAR

PUSH@ ASCIIZ PathName      ; File/directory name
PUSH  WORD   PathInfoLevel  ; Data required
PUSH@ OTHER  PathInfoBuf    ; Data buffer
PUSH  WORD   PathInfoBufSize ; Data buffer size
PUSH  DWORD  0              ; Reserved (must be zero)
CALL  DosQPathInfo
```

Where

PathName is the ASCIIZ full path name of the file or subdirectory.

PathInfoLevel is the level of path information required.

Path information, where applicable, is based on the most recent **DosClose**, **DosSetFileInfo**, or **DosSetPathInfo**.

Level 0x0001 information is returned in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
};
```

Level 0x0002 simply adds cbList to level 0x0001, and uses the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
    long            cbEOF;
    long            cbAlloc;
    unsigned short  attr;
    unsigned short  cbList;
};
```

Level 0x0003 path information returns a subset of the EA information for the file. On input, PathInfoBuf is an EAOP structure above. fpGEAList points to a GEA list defining the attribute names whose values will be returned. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, PathInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR_BUFFER_OVERFLOW) cbList will still be valid, assuming there's at least enough space for it. Its value will be the size of the entire EA set for the file, even though only a subset of attributes was requested.

Level 0x0004 path information returns all EA info for the file. On input, PathInfoBuf is an EAOP structure above. fpGEAList contents are ignored. fpFEAList points to a data area where the relevant FEA list will be returned. The length field of this FEA list is valid, giving the size of the FEA list buffer. offError is ignored.

On output, PathInfoBuf is unchanged. The buffer pointed to by fpFEAList is filled in with the returned information.

If the buffer fpFEAList points to isn't large enough to hold the returned information (ie ERROR_BUFFER_OVERFLOW) cbList will still be valid.

Level 0x0005 path information returns the fully qualified (true) ASCIIZ name of **PathName** in **FileInfoBuf**.

Level 0x0006 requests a file system to verify the correctness of **PathName** per its rules of syntax. An erroneous name is indicated by an error return-code.

PathInfoBuf is the storage area where the system returns the requested level of path information.

PathInfoBufSize is the length of **PathInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR_PATH_NOT_FOUND - a component of PathName is not found.
- ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.
- ERROR_INVALID_LEVEL - the specified FileInfoLevel is not supported.
- Device-driver/device-manager [errors listed](#).
- ERROR_INVALID_EA_NAME - there is an illegal character in an EA name. The offError field points to the offending GEA.
- ERROR_EA_LIST_INCONSISTENT - the cbList does not match the sum of the lengths of the GEA structures.

DosQSysInfo - Query System Information

Purpose

Query static system variables

Format

Calling Sequence:

EXTRN DosQSysInfo:FAR

```
PUSH WORD Index          ; Which variable
PUSH@ OTHER DataBuf      ; System info buffer
PUSH WORD DataBufLen     ; Data buffer size
CALL DosQHandType
```

Where

Index is the ordinal of the system variable to return.

- Index == 0 indicates maximum path length. The maximum path length will be returned in the first word of the DataBuf.

DataBuf is where the system returns the variable value.

DataBufLen is the length of the data buffer.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_PARAMETER - the index is invalid.
- ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.

DosQVerify - Query Verify Setting

Purpose

Returns the value of the Verify flag.

Format

Calling Sequence:

```
EXTRN DosQVerify:FAR
```

```
PUSH@ WORD VerifySetting ; Verify setting
CALL  DosQVerify
```

Where

VerifySetting is the current Verify mode for the requesting process

- If value = 0x00 is returned, verify mode is not active
- If value = 0x01 is returned, verify mode is active

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- none

Remarks

DosRead - Read from a File

Purpose

Reads the specified number of bytes from a file to a buffer location.

Format

Calling Sequence:

```
EXTRN DosRead:FAR
```

```
PUSH  WORD  FileHandle    ; File Handle
PUSH@ OTHER BufferArea    ; Address of user buffer
PUSH  WORD  BufferLength   ; Buffer length
PUSH@ WORD  BytesRead     ; Bytes read
CALL  DosRead
```

Where

FileHandle is the 2-byte file handle obtained from **DosOpen**.

BufferArea is address of the input buffer.

BufferLength is the number of bytes to be read.

BytesRead is where the system returns the number of bytes read.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the handle is not in use or is attached to a physical device.
- **ERROR_ACCESS_DENIED** - the handle is opened as write-only.
- **ERROR_LOCK_VIOLATION** - the region of the file is partially or completely locked by another process.
- Device-driver/device-manager [errors listed](#).

Remarks

It is not guaranteed that the requested number of bytes will be read.

A **BufferLength** value of zero is not considered an error. In the case when the value of **BufferLength** is zero, the system treats the request as a null operation.

Named Pipe Considerations

Reads bytes or messages from a pipe. There are three cases:

1. Byte pipe. The pipe must be in byte read mode (an error is returned if in message read mode). All currently available data, up to the size requested, is returned.
2. Message pipe, message read mode. A read that is larger than the next available message returns only that message and **BytesRead** is set to indicate the size of the message returned. A read that is smaller than the next available message returns with the number of bytes requested and a MORE DATA error code. Subsequent **DosReads** will continue reading the message. **DosPeekNmPipe** may be used to determine how many bytes are left in the message.
3. Message pipe, byte read mode. Reads the pipe as if it were a byte stream, skipping over message headers. This is like reading a byte pipe in byte mode.

When blocking mode is set, a read blocks until data is available. In this case, the read will never return with **BytesRead**=0 except at EOF. Note that in message read mode, messages are always read in their ENTIRETY, except in the case where the message is bigger than the size of the read.

When nonblocking mode is set, a read will return with **BytesRead**=0 at EOF. An error will be returned if there is no data available.

Note: when resuming the reading of a message after a **MORE DATA** indication, the reads will always block until the next piece (or rest) of the message can be transferred. Non-blocking mode only allows a return with **BytesRead**=0 when trying to read at the start of a message and no message is available.

DosReadAsync - Async Read from a File

Purpose

Transfers the specified number of bytes from a handle to a buffer location, asynchronously with respect to the

requesting process's execution.

Format

Calling Sequence:

```
EXTRN DosReadAsync:FAR

PUSH WORD FileHandle ; File handle
PUSH@ DWORD RamSemaphore ; Address of Ram semaphore
PUSH@ WORD ReturnCode ; Address of I/O error RC
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesRead ; Bytes read
CALL DosReadAsync
```

Where

FileHandle is the 2-byte file handle obtained from **DosOpen**.

RamSemaphore is used by the system to post operation complete to the caller.

ReturnCode is where the system returns the I/O operation return code.

BufferArea is address of the input buffer.

BufferLength is the number of bytes to be read.

BytesRead is where the system returns the number of bytes read.

Returns:

AX = 0

ReturnCode = Error code if error

AX = Error Code:

- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR_ACCESS_DENIED - the handle is opened as write-only.
- ERROR_LOCK_VIOLATION - the region of the file is partially or completely locked by another process.
- Device-driver/device-manager [errors listed](#).

Remarks

It is not guaranteed that the requested number of bytes [will be read](#).

A **BufferLength** value of zero is not considered an error. In the case when the value of **BufferLength** is zero, the system treats the request as a null operation.

RamSemaphore must be set by the application before the **DosReadAsync** call is made. The application issues the following sequence:

- DosSemSet ...
- DosReadAsync ...
- DosSemWait ...

Named Pipe Considerations

Reads bytes or messages from a pipe. There are three cases:

1. Byte pipe. The pipe must be in byte read mode (an error is returned if in message read mode). All currently available data, up to the size requested, is returned.
2. Message pipe, message read mode. A read that is larger than the next available message returns only that message and **BytesRead** is set to indicate the size of the message returned. A read that is smaller than the next available message returns with the number of bytes requested and a MORE DATA error code. Subsequent DosReads will continue reading the message. DosPeekNmPipe may be used to determine

how many bytes are left in the message.

3. Message pipe, byte read mode. Reads the pipe as if it were a byte stream, skipping over message headers. This is like reading a byte pipe in byte mode.

When blocking mode is set, a read blocks until data is available. In this case, the read will never return with BytesRead=0 except at EOF. Note that in message read mode, messages are always read in their ENTIRETY, except in the case where the message is bigger than the size of the read.

When nonblocking mode is set, a read will return with BytesRead=0 if no data is available at the time of the read.

Note: when resuming the reading of a message after a **MORE DATA** indication, the reads will always block until the next piece (or rest) of the message can be transferred. Non-blocking mode only allows a return with BytesRead=0 when trying to read at the start of a message and no message is available.

DosRmdir - Remove Subdirectory

Purpose

Removes a subdirectory from the specified disk.

Format

Calling Sequence:

```
EXTRN DosRmdir:FAR
```

```
PUSH@ ASCIIZ DirName ; Directory name
PUSH  DWORD  0       ; Reserved (must be zero)
CALL  DosRmdir
```

Where

DirName is the ASCIIZ directory path name.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR_PATH_NOT_FOUND - the drive is invalid or a directory in the path is not found.
- ERROR_CURRENT_DIRECTORY - the specified path is the current directory of a process.
- ERROR_ACCESS_DENIED - the root directory is specified, or the directory is not empty.
- Device-driver/device-manager [errors listed](#).

Remarks

The directory must be empty before it can be removed with the exception of the "." and "..". You cannot remove subdirectories that contain hidden files. The last directory name in the path is the directory to be removed. The root directory and the current directory cannot be removed.

DosScanEnv - Scan Environment Segment

Purpose

Scans an environment segment for an environment variable.

Format

Calling Sequence:

EXTRN DosScanEnv:FAR

```
PUSH@ ASCIIZ EnvVarName    ; Environment variable name
PUSH@ DWORD ResultPointer ; Search result pointer
CALL DosScanEnv
```

Where

EnvVarName points to the ASCIIZ name of the environment variable of interest. Do not include a trailing "=", since this is not part of the name.

ResultPointer is where the system returns the pointer to the environment string. **ResultPointer** points to the first character of the string which is the value of the environment variable and can be passed directly into **DosSearchPath**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_ENVVAR_NOT_FOUND** - the specified environment variable is not found in the environment segment.

Remarks

Assume that the process's environment contains:

```
"DPATH=c:\sysdir;c:\libdir"
  A
  |
+---- ResultPointer points here after call
      to DosScanEnv below.
```

```
DosScanEnv("DPATH", &ResultPointer);
```

As noted above, **ResultPointer** will point to the first character of the value of the environment variable.

DosSearchPath - Search a path for a file name

Purpose

Provides a general path search mechanism which allows applications to find files residing along paths. The path string may come from the process's Environment, or be supplied directly by the caller.

Format

Calling Sequence:

```

EXTRN DosSearchPath:FAR

PUSH WORD Control ; Function control vector
PUSH@ ASCIIZ PathRef ; Search path reference
PUSH@ ASCIIZ FileName ; File name
PUSH@ OTHER ResultBuffer ; Search result buffer
PUSH WORD ResultBufferLen ; Search result buffer length
CALL DosSearchPath

```

Where

Control is a word bit vector which controls the behaviour of **DosSearchPath**

- Bit 0 = Implied Current Bit
- Bit 1 = Path Source Bit
- Bits 2-15 = Reserved bits, must be 0.

The Implied Current Bit controls whether the current directory is implicitly on the front of the search path. If the Implied Current Bit = 0, **DosSearchPath** will only search the current directory if it appears in the search path. If the Implied Current Bit = 1, **DosSearchPath** will search the current working directory before it searches the directories in the search path.

Implied Current Bit = 0 and Path = ".;a;b"

is equivalent to

Implied Current Bit = 1 and Path = "a;b"

The Path Source Bit determines how **DosSearchPath** interprets the **PathRef** argument. If the Path Source Bit = 0, then **PathRef** points to the actual search path. The search path string may be anywhere in the calling process's address space, hence it may be in the environment, but does not have to be. If the Path Source Bit = 1, then **PathRef** points to the name of an environment variable in the process's environment, and that environment variable contains the search path.

PathRef points to an ASCIIZ string.

If the Path Source Bit of **Control** = 0, then **PathRef** points directly to the search path, which may be anywhere in the caller's address space.

If the Path Source Bit of **Control** = 1, then **PathRef** points to a string which is the name of an environment variable which contains the search path.

A search path consists of a sequence of paths separated by ";". It is a single ASCIIZ string. The directories will be searched in the order they appear in the path.

Environment variable names are simply strings which match name strings in the environment. The "=" sign is not part of the name.

FileName points to the ASCIIZ file name to search for. It may contain wild cards. If **FileName** does contain wild cards, they will remain in the result path returned in **ResultBuffer**. This allows applications like cmd.exe to feed the output directly to **DosFindFirst**. If there are no wildcards in **FileName**, the result path returned in **ResultBuffer** will be a full qualified name, and may be passed directly to **DosOpen**, or any other system call.

ResultBuffer holds the result pathname of the file, if found. If **FileName** is found in one of the directories along the path, its full pathname will be returned in **ResultBuffer**. (With wildcards from **FileName** left in place.) Do not depend on the contents of **ResultBuffer** being meaningful if **DosSearchPath** doesn't return with AX=0.

ResultBufLen is the length of **ResultBuffer**, in bytes.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.
- ERROR_FILE_NOT_FOUND - no matching file is found.
- Device-driver/device-manager [errors listed](#).

Remarks

PathRef always points to an ASCII string.

example:

Let **DPATH** be an environment variable in the environment segment of the process.

```
"DPATH=c:\sysdir;c:\init"    /* in the environment */
```

The following two code fragments are equivalent:

```
DosScanEnv("DPATH", &PathRef);
DosSearchPath(0, /* Path Source Bit = 0 */
    PathRef, "myprog.ini", &ResultBuffer, ResultBufLen);

DosSearchPath(2, /* Path Source Bit = 1 */
    "DPATH", "myprog.ini", &ResultBuffer, ResultBufLen);
```

Both of them use the search path stored as **DPATH** in the environment segment. In the first case, the application uses **DosScanEnv** to find the variable, in the second case **DosSearchPath** calls **DosScanEnv** for the application.

DosSearchPath does no consistency checking or formatting on the names, it simply does a **DosFindFirst** on a series of names it constructs from **PathRef** and **FileName**. This means that the underlying file system does the name formatting.

DosSelectDisk - Select Default Drive

Purpose

Selects the drive specified as the default drive for the calling process.

Format

Calling Sequence:

```
EXTRN DosSelectDisk:FAR

PUSH  WORD DriveNumber      ; Default drive number
CALL  DosSelectDisk
```

Where

DriveNumber contains the default drive number (1=A, 2=B, ...).

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_DRIVE - the specified drive is invalid.

DosSetFHandState - Set File Handle State

Purpose

Set the state of the specified file.

Format

Calling Sequence:

```
EXTRN DosSetFHandState:FAR
```

```
PUSH WORD FileHandle ; File handle
```

```
PUSH WORD FileHandleState ; File handle state
```

```
CALL DosSetFHandState
```

Where

FileHandle is the handle of the file to be set.

FileHandleState is the file handle state and consists of the following bit fields. They are the:

- Inheritance flag
- Write-through flag
- Fail-errors flag
- Zero bit field

File Handle State Bits

```
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0 W F R R R R R I 0 0 0 R 0 0 0
```

I

Inheritance Flag

If I = 0; File handle is inherited by a spawned process resulting from a **DosExecPgm** call.

If I = 1; File handle is private to the current process.

This bit is not inherited by child processes.

W

File Write-through

The file is opened as follows:

If W = 0; Writes to the file may be run through the DOS buffer cache.

If W = 1; Writes to the file may go through the DOS buffer cache but, the sectors will be written (actual file I/O completed) before a synchronous write call returns. This state of the file defines it as a synchronous file.

This bit is not inherited by child processes.

E

Fail-Errors

Media I/O errors will be handled as follows:

If F = 0; reported through the system
critical error handler.
If F = 1; reported directly to the
caller via return code.

This bit is not inherited by child processes. Media I/O errors generated through an IoCtl Category 8 function, always get reported directly to the caller via return code. The **Fail-Errors** functionality applies only to non-ioctl handle-based type file I/O calls.

Q

Zero bits

These bits must be set to zero.

Any other values for **FileHandleState** are invalid.

R

Reserved bits

These bits are reserved and should be set to the values returned by **DosQFHandState** in these positions.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR_INVALID_PARAMETER - the specified mode contains reserved fields non-zero or contains an invalid value.

Remarks

Setting the write-through flag does not affect any previous writes which may have been done. That data may remain in the buffers.

The file handle state bits set by this function can be queried with the **DosQFHandState** function call.

Named Pipe Considerations

Allows setting of the inheritance (I) and write-through (W) bits. Note that setting W to 1 prevents write-behind operations on remote pipes.

DosSetFileInfo - Set a File's Information

Purpose

Specifies information for a file.

Format

Calling Sequence:

EXTRN DosSetFileInfo:FAR

```
PUSH WORD FileHandle      ; File handle
PUSH WORD FileInfoLevel    ; File info data type
PUSH@ OTHER FileInfoBuf    ; File info buffer
PUSH WORD FileInfoBufSize  ; File info buffer size
CALL DosSetFileInfo
```

Where

FileHandle is the file handle.

FileInfoLevel is the level of file information being defined.

Level 0x0001 file information is set from **FileInfoBuf** in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
};
```

Level 0x0002 file information sets a series of EA name/value pairs. On input, **FileInfoBuf** is an EAOP structure above. **fpGEAList** is ignored. **fpFEAList** points to a data area where the relevant FEA list is to be found. **offError** is ignored.

On output, **fpGEAList** is unchanged. **fpFEAList** is unchanged as is the area pointed to by **fpFEAList**. If an error occurred during the set, **offError** will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, **offError** is undefined.

FileInfoBuf is the storage area where the system gets the file information.

FileInfoBufSize is the length of **FileInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_LEVEL** - the specified **FileInfoLevel** is not supported.
- **ERROR_INSUFFICIENT_BUFFER** - the specified buffer is too short to contain the stated level of information.
- **ERROR_DIRECT_ACCESS_HANDLE** - the handle is opened with the direct-open bit set.
- **ERROR_INVALID_PARAMETER** - invalid times/dates are specified.
- Device-driver/device-manager [errors listed](#).
- **ERROR_INVALID_EA_NAME** - there is an illegal character in an EA name. The **offError** field points to the offending FEA.
- **ERROR_EA_LIST_INCONSISTENT** - the **cbList** does not match the sum of the lengths of the FEA structures.
- **ERROR_EA_VALUE_UNSUPPORTED** - the FSD detects an error in the value portion of an EA. The **offError** field points to the offending FEA.

Remarks

The **DosSetFileInfo** level 0x0001 structure is a prefix of the **DosQFileInfo** level 0x0001 structure.

DosSetFileInfo will work only for files opened in a mode that allows write-access.

A zero (0) value in both the date and time components of a field will cause that field to be left unchanged. For example, if both 'Last write date' and 'Last write time' are specified as zero in the Level 0x0001 information structure, then both attributes of the file are left unchanged. If either 'Last write date' or 'Last write time' are specified as non-zero, then both attributes of the file will be set to the new values.

The FAT file system supports only the modification date and time. Creation and access dates and times will not be affected.

DosSetFileMode - Set File Mode

Purpose

Change the mode (attribute) of the specified file.

Format

Calling Sequence:

```
EXTRN DosSetFileMode:FAR

PUSH@ ASCIIZ FileName      ; File path name
PUSH  WORD  NewAttribute    ; New attribute of file
PUSH  DWORD 0               ; Reserved (must be zero)
CALL  DosSetFileMode
```

Where

FileName is the ASCIIZ file path name.

NewAttribute is the file's new attribute.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_PATH_NOT_FOUND - a component of the directory path is not present.
- ERROR_FILE_NOT_FOUND - the last component of the file name is not present on the media.
- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR_ACCESS_DENIED - the attributes are invalid or the specified file name is a character device.
- ERROR_SHARING_BUFFER_EXCEEDED - there is not enough memory to hold sharing information.
- ERROR_SHARING_VIOLATION - the file is currently in use by another process.
- Device-driver/device-manager [errors listed](#).

Remarks

Attributes for Volume Label (0x0008) or Subdirectory (0x0010) cannot be set using **DosSetFileMode**. If the above referenced attributes are used to change a file's mode, an error code is returned.

File attributes are defined as follows.

<u>0x0001</u>	read only file.
<u>0x0002</u>	hidden file.
<u>0x0004</u>	system file.
<u>0x0008</u>	volume label.
<u>0x0010</u>	subdirectory.
<u>0x0020</u>	file archive.

All other attributes are reserved.

DosSetFsInfo - Set File System Information

Purpose

Set information for a file system device.

Format

Calling Sequence:

EXTRN DosSetFsInfo:FAR

```
PUSH WORD DriveNumber      ; Drive number
PUSH WORD FSInfoLevel      ; File system data type
PUSH@ OTHER FSInfoBuf      ; File system info buffer
PUSH WORD FSInfoBufSize    ; File system info buffer size
CALL DosSetFsInfo
```

Where

DriveNumber is the logical drive number (0=default, 1=A,2=B,3=C, ...) and represents the FSD for the media currently in that drive. A value of '0xFFFF' notes that **FSInfoBuf** contains the ASCIIZ path name of the FSD.

FSInfoLevel is the level of file information to set.

Level 0x0001 is reserved.

Level 0x0002 information is defined in the following format:

```
struct {
    unsigned char cbVolLabel;
    unsigned char szVolLabel[];
};
```

cbVolLabel

length of volume label

szVolLabel

asciiz volume label.

FSInfoBuf is where the system gets the new file system information.

FSInfoBufSize is the length of **FSInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_LEVEL** - the specified FSInfoLevel is not supported.
- **ERROR_INVALID_DRIVE** - the specified drive is invalid.
- **ERROR_CANNOT_MAKE** - cannot create volume label
- **ERROR_INVALID_NAME** - there are characters in the volume label that are illegal.
- **ERROR_INSUFFICIENT_BUFFER** - the specified buffer is too short to contain the stated level of information.
- **ERROR_LABEL_TOO_LONG** - the volume label exceeded the file systems name capacity.
- Device-driver/device-manager [errors listed](#).

Remarks

Trailing blanks supplied at volume label definition time are not returned by **DosQFInfo**.

DosSetMaxFH - Set Maximum File Handles

Purpose

This function call defines the maximum number of file handles for the current process.

Format

Calling Sequence:

EXTRN DosSetMaxFH:FAR

PUSH WORD NumberHandles ; Number of file handles
CALL DosSetMaxFH

Where

NumberHandles is the total number of file handles to be provided.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_NOT_ENOUGH_MEMORY** - unable to allocate storage for new file handle table.
- **ERROR_INVALID_PARAMETER** - the requested number of file handles is > 32767 or the requested number is smaller than the currently allocated set.

Remarks

All currently open file handles are preserved.

DosSetPathInfo - Set a File's or Directory's Information

Purpose

Specifies information for a file or a directory.

Format

Calling Sequence:

EXTRN DosSetPathInfo:FAR

PUSH@ ASCIIZ PathName ; File/dir full name
PUSH WORD PathInfoLevel ; Info data type
PUSH@ OTHER PathInfoBuf ; Info buffer
PUSH WORD PathInfoBufSize ; Info buffer size

```
PUSH    DWORD    0                ; Reserved (must be zero)
CALL    DosSetPathInfo
```

Where

PathName is the ASCIIZ full path name of the file or subdirectory.

PathInfoLevel is the level of file/directory information being defined.

Level 0x0001 file information is in the following format:

```
struct {
    unsigned short  dateCreate;
    unsigned short  timeCreate;
    unsigned short  dateAccess;
    unsigned short  timeAccess;
    unsigned short  dateWrite;
    unsigned short  timeWrite;
};
```

Level 0x0002 file information sets a series of EA name/value pairs. On input, PathInfoBuf is an EAOP structure above. fpGEAList is ignored. fpFEAList points to a data area where the relevant FEA list is to be found. offError is ignored.

On output, fpGEAList is unchanged. fpFEAList is unchanged as is the area pointed to by fpFEAList. If an error occurred during the set, offError will be the offset of the FEA where the error occurred. The API return code will be the error code corresponding to the condition generating the error. If no error occurred, offError is undefined.

PathInfoBuf is the storage area where the system gets the file information.

PathInfoBufSize is the length of **PathInfoBuf**.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_LEVEL - the specified PathInfoLevel is not supported.
- ERROR_FILENAME_EXCED_RANGE - the path specified is unacceptable to the FSD managing the volume.
- ERROR_INSUFFICIENT_BUFFER - the specified buffer is too short to contain the stated level of information.
- Device-driver/device-manager [errors listed](#).
- ERROR_INVALID_EA_NAME - there is an illegal character in an EA name. The offError field points to the offending FEA.
- ERROR_EA_LIST_INCONSISTENT - the cbList does not match the sum of the lengths of the FEA structures.
- ERROR_EA_VALUE_UNSupportable - the FSD detects an error in the value portion of an EA. The offError field points to the offending FEA.

Remarks

With respect to files, **DosSetPathInfo** will work only for closed files.

A zero (0) value in both the date and time components of a field will cause that field to be left unchanged. For example, if both 'Last write date' and 'Last write time' are specified as zero in the Level 0x0001 information structure, then both attributes of the file are left unchanged. If either 'Last write date' or 'Last write time' are specified as non-zero, then both attributes of the file will be set to the new values.

DosSetVerify - Set/Reset Verify Switch

Purpose

Sets the verify switch.

Format

Calling Sequence:

```
EXTRN DosSetVerify:FAR

PUSH WORD VerifySetting ; New value of verify switch
CALL DosSetVerify
```

Where

VerifySetting is the new state of Verify Mode for the requesting process

- If value = 0 is specified, Verify Mode is deactivated
- If value = 1 is specified, Verify Mode is activated

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- ERROR_INVALID_VERIFY_SWITCH

Remarks

When verify is on, device drivers are requested to perform a verify-after write operation each time they do a file write to assure proper data recording on the disk.

DosWrite - Synchronous Write to a File

Purpose

Transfers the specified number of bytes from a buffer to the specified file, synchronously with respect to the requesting process's execution.

Format

Calling Sequence:

```
EXTRN DosWrite:FAR

PUSH WORD FileHandle ; File handle
PUSH@ OTHER BufferArea ; Address of user buffer
PUSH WORD BufferLength ; Buffer length
PUSH@ WORD BytesWritten ; Bytes written
CALL DosWrite
```

Where

FileHandle is the 2-byte file handle obtained from **DosOpen**.

BufferArea is address of the output buffer.

BufferLength is the number of bytes to be written.

BytesWritten is where the system returns the number of bytes written.

Returns:

IF ERROR (AX not = 0)

AX = Error Code:

- **ERROR_INVALID_HANDLE** - the handle is not in use or is attached to a physical device.
- **ERROR_LOCK_VIOLATION** - the region of the file is partially or completely locked by another process.
- **ERROR_ACCESS_DENIED** - the file is opened as read-only.
- Device-driver/device-manager [errors listed](#).

Remarks

Upon return from this function, **BytesWritten** is the number of bytes **actually** written.

A **BufferLength** value of zero is not considered an error. In the case when the value of **BufferLength** is zero, the system treats the request as a null operation.

Named Pipe Considerations

Writes bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; DosWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Writes in blocking mode always write all requested bytes before returning. In nonblocking mode, writes return either with all bytes written or none written; the latter will occur in certain cases where the DosWrite would have to block in order to complete the request (e.g., no room in pipe buffer or buffer currently being written by another client).

An attempt to write to a pipe whose other end has been closed will return with **ERROR_BROKEN_PIPE**.

DosWriteAsync - Asynchronous Write to a File

Purpose

Transfers the specified number of bytes to a handle from a buffer location, asynchronously with respect to the requesting process's execution.

Format

Calling Sequence:

EXTRN DosWriteAsync:FAR

```
PUSH WORD    FileHandle    ; File handle
PUSH@ DWORD  RamSemaphore  ; Address of Ram semaphore
PUSH@ WORD   ReturnCode    ; Address of I/O error RC
PUSH@ OTHER  BufferArea     ; Address of user buffer
PUSH WORD    BufferLength   ; Buffer length
PUSH@ WORD   BytesWritten   ; Bytes written
CALL DosWriteAsync
```

Where

FileHandle is the 2-byte file handle obtained from **DosOpen**.

RamSemaphore is used by the system to post operation complete to the caller.

ReturnCode is where the system returns the I/O operation return code.

BufferArea is address of the output buffer.

BufferLength is the number of bytes to be written.

BytesWritten is where the system returns the number of bytes written.

Returns:

AX = 0

ReturnCode = Error code if error

Error Codes:

- ERROR_INVALID_HANDLE - the handle is not in use or is attached to a physical device.
- ERROR_LOCK_VIOLATION - the region of the file is partially or completely locked by another process.
- ERROR_ACCESS_DENIED - the file is opened as read-only.
- Device-driver/device-manager [errors listed](#).

Remarks

Upon return from this function, **BytesWritten** is the number of bytes **actually** written.

A **BufferLength** value of zero is not considered an error. In the case when the value of **BufferLength** is zero, the system treats the request as a null operation.

RamSemaphore must be set by the application before the **DosWriteAsync** call is made. The application issues the following sequence:

- DosSemSet ...
- DosWriteAsync ...
- DosSemWait ...

Named Pipe Considerations

Writes bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; DosWrite automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Writes in blocking mode always write all requested bytes before returning. In nonblocking mode, writes return either with all bytes written or none written; the latter will occur in certain cases where the DosWrite would have to block in order to complete the request (e.g., no room in pipe buffer or buffer currently being written by another client).

An attempt to write to a pipe whose other end has been closed will return with ERROR_BROKEN_PIPE.

FSD System Interfaces

This page is intentionally left blank.

Overview of Installable File System Driver Dispatch

Notes:

1. Installable file system entry points are called by the kernel as a result of action taken through the published standard file I/O application programming interface in the 1.3 version of the OS/2 Operating System and Utilities Functional Specification.
2. Installable file systems will be installed as OS/2 dynamic link library modules. Unlike device drivers, they may include any number of segments, all of which will remain after initialization unless the FSD itself takes some action to free them.
3. An FSD will export FS entries to the kernel using standard public declarations. Each FS entry will be called directly. The kernel will manage the association between internal data structures and FSDs.
4. When a file system service is required, OS/2 will assemble an argument list, and call the appropriate FS entry for the relevant FSD. If a back-level FSD is loaded, the kernel will assure that all arguments passed and all structures passed are understood by the FSD.
5. Application program interfaces that are unsupported by an FSD, must receive `ERROR_UNSUPPORTED_FUNCTION` from the FSD.

Data Structures

As mentioned previously, OS/2 data structures will need to be expanded to include a pointer to the file system driver. The affected structures include the CDS (current directory structure), the SFT (system file table entry), the VPB (volume parameter block), and the file search structures. In addition, these structures need to include areas which are defined to be file system specific.

The file system service routines will generally be passed pointers to two parameter areas in addition to read-only parameters which will be specific to each call. The FSD does not need to verify these pointers. The two parameter areas will contain file system independent data which are maintained jointly by the OS/2 and the file system driver and an area of file system dependent data which will be unused by OS/2 and which may be used in any way the file system driver wishes. The file system driver is generally permitted to use the file system dependent information in any way it sees fit; it may contain all the information needed to describe the current state of the file or directory, or it may contain a 'handle' which will direct it to other information about the file maintained within the FSD.

The following definitions are used in the file system service routine definitions in the following section for the file system dependent and file system independent parameter areas.

Disk media and file system layout are described by the following structures. The data which are provided to the file system may depend on the level of file system support provided by the device driver attached to the block device. These structures are relevant only for local file systems.

```
/* file system independent - volume params */
struct vpfsi {
    unsigned long   vpi_vid;           /* 32 bit volume ID */
    unsigned long   vpi_hDEV;          /* handle to device driver */
    unsigned short  vpi_bsize;         /* sector size in bytes */
    unsigned long   vpi_totsec;        /* total number of sectors */
    unsigned short  vpi_trksec;        /* sectors / track */
    unsigned short  vpi_nhead;        /* number of heads */
    char            vpi_text[12];      /* asciiz volume name */
}; /* vpfsi */

/* file system dependent - volume params */
struct vpfsd {
    char            vpd_work[36];      /* work area */
}; /* vpfsd */
```

Per-disk current directories are described by the following structures. These structures can only be modified by the FSD during `FS_ATTACH` and `FS_CHDIR` operations.

```
/* file system independent - current dirs */
struct cdfsi {
    unsigned long   cdi_hVPB;          /* VPB handle for associated device */
    unsigned short  cdi_end;           /* offset to root of path */
};
```

```

        char          cdi_flags;          /* fs independent flags */
        char          cdi_curdir[MAXPATHLEN]; /* text of current directory */
}; /* cdfsi */

/* file system dependent - current dirs */
struct cdfsd {
    char          cdd_work[8];          /* work area */
}; /* cdfsd */

```

Open files are described by data initialized at file open time and discarded at the time of last close of all file handles which had been associated with that open instance of that file. There may be multiple open file references to the same file at any one time. It is the responsibility of the FSD to assure that any changes to the file times and standard OS/2 attributes are reflected in that file's SFT. OS/2 will distribute the changes to other open file instances.

FSDs are required to support direct access opens. These are indicated by a bit set in the sffsi.sfi_mode field.

```

/* file system independent - file instance */
struct sffsi {
    unsigned long  sfi_mode;          /* access/sharing mode */
    unsigned long  sfi_hVPB;          /* volume info. */
    unsigned short sfi_ctime; /* file creation time */
    unsigned short sfi_cdate; /* file creation date */
    unsigned short sfi_atime; /* file access time */
    unsigned short sfi_adate; /* file access date */
    unsigned short sfi_mtime; /* file modification time */
    unsigned short sfi_mdate; /* file modification date */
    unsigned long  sfi_size; /* size of file */
    unsigned long  sfi_position; /* read/write pointer */
/* the following may be of use in sharing checks */
    unsigned long  sfi_UID; /* user ID of initial opener */
    unsigned long  sfi_PID; /* process ID of initial opener */
    unsigned long  sfi_PDB; /* PDB (in 3.x box) of initial opener */
    unsigned long  sfi_selfsfn; /* system file number of file instance */
}; /* sffsi */

/* file system dependent - file instance */
struct sffsd {
    char          sfd_work[30]; /* work area */
}; /* sffsd */

```

The Program Data Block or program header, PDB (sfi_pdb), is the unit of sharing for 3.X Box processes. For protect mode processes, the unit of sharing is the Process ID, PID (sfi_pid). FSDs should use the combination <PDB, PID, UID> as indicating a distinct process.

File search records in OS/2 are constrained to fit within the 21 byte reserved area defined for FindFirst and FindNext due to DOS 3 and Network compatibility issues. These are the areas passed to the FSD.

```

/* file system independent - file search parameters */
struct fsfsi {
    unsigned long  fsi_hvpb;          /* volume info. */
}; /* fsfsi */

/* file system dependent - file search parameters */
struct fsfsd {
    char          fsd_work[16]; /* work area */
}; /* fsfsd */

```

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, may not necessarily support all the described information kept on a file basis. When such is the case, file system drivers are required to return to the application a null (zero) value for the unsupported parameter (when the unsupported data are a subset of the data returned by the API) or to return ERROR_NOT_SUPPORTED (when all of the data returned by the API is unsupported).

FSD calling conventions and requirements

Calling conventions between FS router, FSD, and FS helpers are:

- Arguments will be pushed in left-to-right order onto the stack.
- The callee is responsible for cleaning up the stack.
- Registers DS, EBX, ESI, EDI, EBP, SS, ESP are preserved.
- Return conditions appear in EAX with the convention that EAX == 0 indicates successful completion. EAX != 0 indicates an error with the value of EAX being the error code.

Interrupts must ALWAYS be enabled and the direction flag should be presumed to be undefined; calls to the FS helpers will change the direction flag at will.

In OS/2, file system drivers will always be called in kernel protect mode. This has the advantage of allowing the FSD to execute code without having to account for preemption; no preemption occurs when in kernel mode. While this greatly simplifies FSD structure, it forces the FSD to yield the CPU when executing long segments of code. In particular, an FSD must not hold the CPU for more than 2 milliseconds at a stretch. The FSD helper FSH_YIELD is provided so that FSDs may relinquish the CPU.

The file system drivers cannot have any interrupt time activations. Since they occupy high, movable, and swappable memory, there is no guarantee upon addressability of the memory at interrupt time.

Each FS service routine may block.

Error codes

The FSD should use existing error codes when possible. New error codes must be in the range reserved for FSDs. The FS_FSCTL interface must support returning information about new error codes.

FS service routine command names

The following table summarizes the commands associated with each FS entry point, and lists the names the FSD must use to export them. Note that names must be in all upper case as required by OS/2 naming convention.

FS entry point	OS/2 API supported
FS_ATTACH	DOSQFSATTACH, DOSFSATTACH
FS_CHDIR	DOSCHDIR, DOSQCURDIR
FS_CHGFILEPTR	DOSCHGFILEPTR
FS_CLOSE	DOSCLOSE
FS_COMMIT	DOSBUFRESET, DOSCLOSE
FS_DELETE	DOSDELETE
FS_EXIT	DOSEXIT
FS_FILEATTRIBUTE	DOSQFILEMODE, DOSSETFILEMODE
FS_FILEINFO	DOSQFILEINFO, DOSSETFILEINFO
FS_FILEIO	DOSFILEIO, DOSFILELOCKS
FS_FINDCLOSE	DOSFINDCLOSE
FS_FINDFIRST	DOSFINDFIRST
FS_FINDNEXT	DOSFINDNEXT
FS_FINDNOTIFYCLOSE	DOSFINDNOTIFYCLOSE
FS_FINDNOTIFYFIRST	DOSFINDNOTIFYFIRST
FS_FINDNOTIFYNEXT	DOSFINDNOTIFYNEXT
FS_FLUSHBUF	DOSBUFRESET
FS_FSCTL	DOSFSCTL
FS_FSINFO	DOSQFSINFO, DOSSETFSINFO
FS_INIT	--
FS_IOCTL	DOSDEVICTL
FS_MKDIR	DOSMKDIR
FS_MOVE	DOSMOVE
FS_MOUNT	--
FS_NEWSIZE	DOSNEWSIZE
FS_OPENCREATE	DOSOPEN
FS_PATHINFO	DOSQPATHINFO, DOSSETPATHINFO
FS_PROCESSNAME	--

FS_READ	DOSIREAD
FS_RMDIR	DOSRMDIR
FS_SETSWAP	--
FS_WRITE	DOSIWRITE

The following OS/2 file system API routines are implemented entirely in OS/2 and do not need any services from the FSD: DOSDUPHANDLE, DOSQCURDISK, DOSQFHANDSTATE, DOSQHANDTYPE, DOSQVERIFY, DOSSCANENV, DOSSEARCHPATH, DOSSELECTDISK, DOSSETFHANDSTATE, DOSSETMAXFH, and DOSSETVERIFY.

FS Entry Point Descriptions

Each FS entry point has a distinct parameter list composed of those parameters needed by that particular entry. Parameters include:

- File pathnames
- Current disk/directory information
- Open file information
- Application data buffers
- Descriptions of file extended attributes
- Other parameters specific to an individual call

File system drivers which support hierarchical directory structures must use '\' and '/' as path name component separators. File system drivers which do not support hierarchical directory structures must reject as illegal any use of '\' or '/' in path names. The file names '.' and '..' are reserved for use in hierarchical directory structures for the current directory and the parent of the current directory respectively.

Unless otherwise specified in the descriptions below, data buffers may be accessed without concern for the accessibility of the data. In other words, OS/2 will either check buffers for accessibility and lock them, or transfer them into locally accessible data areas.

Simple parameters will be verified by the IFS router before the FS service routine is called.

FS_ATTACH - Attach or Detach An FSD To A Drive or Device

Purpose

Attach or detach a remote drive or pseudo-device to an FSD.

```
int far pascal FS_ATTACH (flag, pDev, pcdfsd, pParm, pLen)
unsigned long      flag;
char *             pDev;
struct cdfs *      pcdfsd;
char *             pParm;
unsigned long *     pLen;
```

Where

flag

indicates attach vs detach.

flag == 0 requests an attach. The FSD is being called to attach a specified drive or character device.

flag == 1 requests a detach.

flag == 2 requests the FSD to fill in the specified buffer with attachment information.

pDev

pointer to the asciiz text of either the drive (drive-letter followed by a colon) or to the character device (must be \DEV\device) that is being attached/detached/queried. The FSD does not need to verify this pointer.

pcdfsd

pointer to structure of file-system dependent working directory information. When an attach/detach/query of a character device is requested, this pointer is null. When attaching a drive, this structure contains no data and is available for the FSD to store information needed to manage the working directory. All subsequent FSD calls (generated by API calls that reference this drive) are passed a pointer to this structure with contents left as the FSD left them. When detaching or querying a drive, this structure contains the data as the FSD left them.

pParm

address of application parameter area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). When an attach is requested this will point to the API-specified user data block that contains information regarding the attach operation (e.g. passwords). This pointer is null for a detach operation. For a query, the kernel will fill in part of the buffer, adjust the pointer, and call the FSD to fill in the rest. (See structure returned by DosQFsAttach, pParm will point to cbFSADData, the FSD should fill in cbFSADData and rgFSADData.) pParm must be verified, even in the query case.

pLen

pointer to length of the application parameter area. For attach, this points to the length of the application data buffer. For detach, this is null. For query, this is the length of the remaining space in the application data buffer. Upon filling in the buffer, the FSD will set this to the length of the data returned. If the data returned is longer than the data buffer length, the FSD should set this value to be the length of the data that query could return. In this case, the FSD should also return ERROR_BUFFER_OVERFLOW. The FSD does not need to verify this pointer.

Remarks

Local FSDs will never get called with attempts to attach or detach drives or queries about drives.

FS_CHDIR - Change/Verify Directory Path

Purpose

Change or verify the directory path for the requesting process.

```
int pascal FS_CHDIR (flag, pcdfsi, pcdfsd, pDir,
                    iCurDirEnd)
unsigned long      flag;
struct cdfsi *    pcdfsi;
struct cdfsd *    pcdfsd;
char *            pDir;
unsigned long      iCurDirEnd;
```

Where

flag

indicates what action is to be taken on the directory.

flag == 0 indicates that an explicit directory-change request has been made.

flag == 1 indicates that the working directory need to be verified.

flag == 2 indicates that this reference to a directory is being freed.

The flag passed to the FSD will have a valid value.

pcdfsi

pointer to file-system independent working directory structure. For flag == 0, this pointer points to the previous current directory on the drive. For flag == 1, this pointer points to the most-recent working directory on the drive. The cdi_curdir field contains the text of the directory that is to be verified. For flag == 2, this pointer is null. The FSD MUST NEVER modify the cdfsi. The kernel handles all updates.

pcdfsd

pointer to file-system dependent working directory structure. This is a place for the FSD to store information about the working directory. For flag == 0 or 1, this is the information left there by the FSD. The FSD is expected to update this information if the directory exists. For flag == 2, this is the information left there by the FSD.

pDir

pointer to directory text. For flag == 0, this is the pointer to the directory. For flag == 1 or flag == 2, this pointer is null. The FSD does not need to verify this pointer.

iCurDirEnd

index of the end of the current directory in pDir. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the directory text, i.e. a device. This parameter only has meaning for flag == 0.

Remarks

The FSD should cache no information when the directory is the root. Root directories are a special case. They always exist, and never need validation. The kernel does not pass root directory requests to the FSD. And an FSD is not allowed to cache any information in the cdfsd for a root directory. Under normal conditions, the kernel does not save the CDS for a root directory and builds one from scratch when it is needed. (One exception is where a validate cds fails, and the kernel sets it to the root, and zeroes out the cdfsd. This CDS is saved and is cleaned up later.)

The following is information about the exact state of the cdfsi and cdfsd passed to the FSD for each flag value and guidelines about what an FSD should do upon receiving an FS_CHDIR call.

```
if (flag == 0) {    /* Set new Current Directory */
    pcdfsi, pcdfsd = copy of CDS we're starting from; maybe useful
                    as useful as starting point for verification.

    cdfsi contents:

        hVPB - handle of Volume Parameter Block mapped to this
              drive

    end - end of 'root' portion of CurDir

    flags - various flags (indicating state of cdfsd)

        IsValid - cdfsd is unknown format (ignore contents)
                  IsValid == 0x80

        IsRoot - cdfsd is meaningless if CurDir = root
                 (not kept)
                  IsRoot == 0x40

        IsCurrent - cdfsd is known format, but may not be
                    current (medium may have been changed)
                      IsCurrent == 0x20
```

```

text - Current Directory Text

icurdir = if Current Directory is in the path of the the new
          Current Directory, this is the index to the end of
          the Current Directory. If not, then this = -1 (Current
          Directory does not apply).

pDir = path to verify as legal directory.

DO THIS
-----

Validate path named in pDir.
/* This means both that it exists AND that it's a dir.
   pcdfsi, pcdfsd, icurdir give old CDS, which may allow
   optimization */

if (Validate succeeds) {
    If (pDir != ROOT)
        store any cache information in area pointed to by
        pcdfsd.
    else
        do nothing!
        /* area pointed to by pcdfsd will be thrown away, so
        don't bother storing into it */

    return success
}
else
    return failure

/* Kernel will create new CDS using pDir data and pcdfsd data.

If the old CDS is valid, the kernel will take care of
cleaning it up.

The FSD MUST NOT edit any structure other than the *pcdfsd
area, with which it may do as it chooses. */

} /* flag == 0 */

else if (flag == 1) { /* Validate current CDS structure */

    pcdfsi = pointer to copy of cdfs of interest.

    pcdfsd = pointer to copy of cdfs.  Flags in cdfs indicate the
        state of this cdfs.  It may be: (1) completely invalid
        (unknown format), (2) known format, but non-current
        information, (3) completely valid, or (4) all zero (root).

    DO THIS
    -----

    Validate that CDS still describes a legal directory (using
    cdi_text)

    if (valid) {
        update cdfs if necessary
        return success
        /* kernel will copy cdfs into real CDS */
    }
    else {
        if (cdi_isvalid)
            release any resources associated with cdfs
            /* kernel will force Current Directory to root, and
            will zero out cdfs in real CDS */

        return failure
    }

    /* The FSD MUST NOT modify any structure other than the cdfs
    pointed to by pcdfsd. */

}

else if (flag == 2) { /* previous CDS no longer in use; being freed */

    pcdfsd = pointer to copy of cdfs of CDS being freed.

    DO THIS
    -----

    Release any resources associated with the CDS.

```

```

/* For example, if cdfsd (where pcdfsd points) contains a
   pointer to some FSD private structure associated with the
   CDS, that structure should be freed. */

/* Kernel will not retain the cdfsd */
}

```

FS_CHGFILEPTR - Move a file's position pointer

Purpose

Move a file's logical read/write position pointer.

```

int pascal FS_CHGFILEPTR (psffsi, psffsd, offset, type)
struct sffsi *      psffsi;
struct sffsd *      psffsd;
long                offset;
unsigned long        type;

```

Where

psffsi

pointer to file-system independent portion of open file instance. The FSD uses the current file size or sfi_position along with offset and type to compute a new sfi_position. This is updated by the system.

psffsd

pointer to file-system dependent portion of open file instance. The FSD may store or adjust data as appropriate in this structure.

offset

signed offset to be added to the current file size or position to form the new position within the file.

type

indicates base of seek operation. type == 0 indicates seek relative to beginning of file. type == 1 indicates seek relative to current position within the file. type == 2 indicate seek relative to end of file. The value of type passed to the FSD will be valid.

Remarks

The file system may want to take the seek operation as a hint that an I/O operation is about to take place at the new position and initiate a positioning operation on sequential access media or a read-ahead operation on other media.

Some 3xbox programs expect to be able to do a negative seek. OS/2 will pass these requests on to the FSD and will return an error for protect mode negative seek requests. Since a seek to a negative position is effectively a seek to a very large offset, it is suggested that the FSD return end-of-file for subsequent read requests.

FSDs must allow seeks to positions beyond end-of-file.

FS_CLOSE - Close a File

Purpose

Closes the specified file handle.

```
int pascal FS_CLOSE (psffsi, psffsd)
struct sffsi * psffsi;
struct sffsd * psffsd;
```

Where

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

Remarks

Called on last close of a file.

Any reserved resources for this instance of the open file may be released. It may be assumed that all open files will be closed at process termination. That is not to say that this entry point will always be called at process termination; it may not necessarily be the last close of an open file.

FS_COMMIT - Commit a file's buffers to Disk

Purpose

Flush requesting process's cache buffers and update directory information for the file handle.

Parameters

```
int pascal FS_COMMIT (psffsi, psffsd)
struct sffsi * psffsi;
struct sffsd * psffsd;
```

Where

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

Remarks

Called on close of a file if the file had been written and also called via DOSBUFRESET. On the final close of a file, FS_CLOSE will also be called. OS/2 reserves the right to call FS_COMMIT even if no changes have been made to the file.

The FSD should update access and modification times, if appropriate.

Any locally cached information about the file must be output to the media. The directory entry for the file is to be updated from the sffsi and sffsd structures.

FS_DELETE - Delete a File

Purpose

Removes a directory entry associated with a filename.

```
int pascal FS_DELETE (pcdfsi, pcdfsd, pFile, iCurDirEnd)
struct cdfsi * pcdfsi;
struct cdfsd * pcdfsd;
char * pFile;
unsigned long iCurDirEnd;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfsd

pointer to file-system dependent working directory structure.

pFile

pointer to asciiz name of file/directory. The FSD does not need to validate this pointer.

iCurDirEnd

index of the end of the current directory in pFile. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

Remarks

The file(s) specified in filename should be deleted.

The deletion of a file opened in compatibility mode in the 3xbox by the same process requesting the delete is supported. OS/2 will call FS_CLOSE for the file before issuing the call to FS_DELETE.

FS_EXIT - End of process

Purpose

Release FSD resources still held after process termination.

```
void pascal FS_EXIT (uid, pid, pdb);
unsigned long uid;
unsigned long pid;
unsigned long pdb;
```

Where

uid

user id of process. This will be a valid value.

pid

process id of process. This will be a valid value.

pdb

3x box process id of process. This will be a valid value.

Remarks

This call is not needed to release file resources since all files are closed on process termination, but it may be helpful if resources are being held due to unterminated searches (in the case of searches initiated from the 3.x box).

FS_FILEATTRIBUTE - Query/Set File Attribute

Purpose

Query/set the attribute of the specified file.

```
int pascal FS_FILEATTRIBUTE (flag, pcdfsi, pcdfsd,  
                             pName, iCurDirEnd, pAttr)  
  
unsigned long    flag;  
struct cdfsi *   pcdfsi;  
struct cdfsd *   pcdfsd;  
char *           pName;  
unsigned long    iCurDirEnd;  
unsigned long *  pAttr;
```

flag

indicates retrieval of attributes vs setting attributes

flag == 0 indicates retrieving the attribute.

flag == 1 indicates setting the attribute.

All other values reserved.

The value of flag passed to the FSD will be valid.

pcdfsi

pointer to file-system independent working directory structure.

pcdfsd

pointer to file-system dependent working directory structure.

pName

pointer to asciiz name of file/directory. The FSD does not need to validate this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

pAttr

pointer to attribute. For flag == 0, the FSD should store the attribute in the indicated location. For flag == 1, the FSD should retrieve the attribute from this location and set it in the file/directory. The FSD does not need to validate this pointer.

FS_FILEINFO - Query/Set a File's Information

Purpose

Returns information for a specific file.

```
int pascal FS_FILEINFO (flag, psffsi, psffsd,
                        level, pData, cbData)
unsigned long  flag;
struct sffsi * psffsi;
struct sffsd * psffsd;
unsigned long  level;
char *         pData;
unsigned long  cbData;
```

Where

flag

indicates retrieval of information vs setting information.

flag == 0 indicates retrieving information.

flag == 1 indicates setting information.

All other values reserved.

The value of flag passed to the FSD will be valid.

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

level

information level to be returned. Level selects among a series of structures of data to be returned.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). When retrieval (flag == 0) is specified, the FSD will place the information into the buffer. When outputting information to a file (flag == 1), the FSD will retrieve that data from the application buffer.

cbData

length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR_BUFFER_OVERFLOW. For flag == 1, this is the length of data to be applied to the file.

Remarks

The supported information levels are described in the OS/2 1.3 API description.

FS_FILEIO - Multi-function File I/O

Purpose

Perform multiple lock, unlock, seek, read, and write I/O.

```
int pascal FS_FILEIO (psffsi, psffsd,
                      pCmdList, cbCmdList, poError)
struct sffsi *      psffsi;
struct sffsd *      psffsd;
char *              pCmdList;
unsigned long        cbCmdList;
unsigned long *      poError;
```

Where

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

pCmdList

pointer to command list that contains entries indicating what commands will be performed. Each individual operation (CmdLock, CmdUnlock, CmdSeek, CmdIO) is performed as atomic operations until all are complete or until one fails. CmdLock executes a multiple range lock as an atomic operation, CmdUnlock executes a multiple range unlock as an atomic operation. The validity of the user address has not been verified (see FSH_PROBEBUF).

For CmdLock, the command format is:

```
CmdLock Struc
Cmd      dw      0      ; 0 for lock operations
Timeout  dd      ?      ; ms timeout for lock success
LockCnt  dw      ?      ; Number of locks that follow
CmdLock  Ends
```

which is followed by a series of records of the following format:

```
Lock      Struc
Share     dw      ?      ; 0 for exclusive, 1 for read-only access
Start     dd      ?      ; start of lock region
Length    dd      ?      ; length of lock region
Lock      Ends
```

If a lock within a **CmdLock** causes a timeout, none of the other locks within the scope of **CmdLock** are in force since the lock operation is viewed as atomic.

CmdLock.Timeout is the count in milliseconds, until the requesting process is to resume execution if the requested locks are not available. If CmdLock.Timeout == 0, there will be no wait. If CmdLock.Timeout < 0xFFFFFFFF it is the number of milliseconds to wait until the requested locks become available. If CmdLock.Timeout == 0xFFFFFFFF then the thread will wait indefinitely until the requested locks become available.

Lock.Share defines the type of access other processes may have to the file-range being locked. value == 0, other processes have 'No-Access' to the locked range. value == 1, other processes have 'Read-Only' access to the locked range.

For CmdUnlock, the command format is:

```
CmdUnlock Struc
Cmd      dw      1      ; 1 for unlock operations.
UnlockCnt dw      ?      ; Number of unlocks that follow
CmdUnlock Ends
```

which is followed by a series of records of the following format:

```
UnLock   Struc
Start    dd      ?    ; start of locked region
Length   dd      ?    ; length of locked region
UnLock   Ends
```

For CmdSeek, the command format is:

```
CmdSeek   Struc
Cmd        dw      ?    ; 2 for seek operation
Method     dw      ?    ; 0 for absolute,
                        ; 1 for relative to current,
                        ; 2 for relative to EOF.
Position   dd      ?    ; file seek position or delta
Actual     dd      ?    ; actual position seeked to
CmdSeek    Ends
```

For CmdIO, the command format is:

```
CmdIO      Struc
Cmd         dw      ?    ; 3 for read, 4 for write
Buffer@     dd      ?    ; ptr to the data buffer
BufferLen   dd      ?    ; number of bytes requested
Actual      dd      ?    ; number of bytes actually
                        ; transferred
CmdIO       Ends
```

cbCmdList

length in bytes of the command list.

poError

offset within the command list of the command that caused the error. This field only has value when an error occurs. The validity of the user address has not been verified (see FSH_PROBEBUF).

Remarks

This function provides a simple mechanism for combining the following operations into a single request and providing improved performance particularly in a networking environment.

Local file systems will never see this call. The command list will be parsed by the IFS router; the FSD will see only FS_CHGFILEPTR, FS_READ, FS_WRITE calls.

Remote file systems will see this call in its entirety. The atomicity guarantee applies only to the commands themselves and not to the list as a whole.

FS_FINDCLOSE - Directory Read (Search) Close

Purpose

Provides the mechanism for an FSD to release resources allocated on behalf of **FS_FINDFIRST** and **FS_FindNext**.

```
int pascal FS_FINDCLOSE (pfsfsi, pfsfsd)
struct fsfsi * pfsfsi;
struct fsfsd * pfsfsd;
```

Where

pfsfsi

pointer to file-system independent file search structure. The FSD should not update this structure.

pfsfsd

pointer to file-system dependent file search structure. The FSD may use this to store information about continuation of the search.

Remarks

DOSFINDCLOSE has been called on the handle associated with the search buffer. Any file system related information may be released.

If FS_FINDFIRST for a particular search returns an error, an FS_FINDCLOSE for that search will not be issued.

FS_FINDFIRST - Find First Matching File Name

Purpose

Find first occurrence of a file name in a directory.

```
int pascal FS_FINDFIRST (pcdfsi, pcdfsd, pName, iCurDirEnd,
                        attr, pfsfsi, pfsfsd,
                        pData, cbData, pcMatch,
                        level )

struct cdfsi *      pcdfsi;
struct cdfsd *      pcdfsd;
char *              pName;
unsigned long       iCurDirEnd;
unsigned long       attr;
struct fsfsi *      pfsfsi;
struct fsfsd *      pfsfsd;
char *              pData;
unsigned long       cbData;
unsigned long *     pcMatch;
unsigned long       level;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfsd

pointer to file-system dependent working directory structure.

pName

pointer to asciiz name of file/directory. Wildcard characters are allowed only in the last component. The FSD does not need to validate this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

attr

bit field that governs the match. Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. For example, an attribute of system and hidden is passed in. A file with the same name and an attribute of system is found. This file should be returned. A file with the same name and no attributes (a regular file) would also be returned. The attributes read-only and file archive will not be passed in and should be ignored when comparing directory attributes. The value of attr passed to the FSD will be valid.

pfsfsi

pointer to file-system independent file search structure. The FSD should not update this structure.

pfsfsd

pointer to file-system dependent file search structure. The FSD may use this to store information about continuation of the search.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData

length of the application data area in bytes.

pcMatch

pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to validate this pointer.

level

information level to be returned. Level selects among a series of structures of data to be returned. The level passed to the FSD will be valid.

Remarks

Sufficient information to find the next matching directory entry must be saved in the fsfsd structure.

If FS_FINDFIRST for a particular search returns an error, an FS_FINDCLOSE for that search will not be issued.

FS_FINDNEXT - Find Next Matching File Name

Purpose

Find next occurrence of a file name in a directory.

```
int pascal FS_FINDNEXT (pfsfsi, pfsfsd, pData, cbData, pcMatch)
struct fsfsi *    pfsfsi;
struct fsfsd *    pfsfsd;
char *            pData;
unsigned long     cbData;
unsigned long *   pcMatch;
```

Where

pfsfsi

pointer to file-system independent file search structure. The FSD should not update

this structure.

pfsfsd

pointer to file-system dependent file search structure. The FSD may use this to store information about continuation of the search.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData

length of the application data area in bytes.

pcMatch

pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to validate this pointer.

Remarks

Sufficient information to find the next matching directory entry must be saved in the fsfsd structure.

FS_FINDNOTIFYCLOSE - Close Find-Notify Handle

Purpose

Closes the association between a 'Find-Notify' handle and a **DOSFINDNOTIFYFIRST** or **DOSFINDNOTIFYNEXT** function. Provides the mechanism for an FSD to release resources allocated on behalf of **FS_FINDNOTIFYFIRST** and **FS_FINDNOTIFYNEXT**.

```
int pascal FS_FINDNOTIFYCLOSE (handle)
unsigned long handle;
```

Where

handle

directory handle. This handle was returned by the FSD and is associated with a previous FS_FINDNOTIFYFIRST or FS_FINDNOTIFYNEXT call.

Remarks

FS_FINDNOTIFYCLOSE has been called on the handle associated with a FS_FINDNOTIFYFIRST. Any file system related information may be released.

FS_FINDNOTIFYFIRST - Start monitoring directory for changes

Purpose

Start monitoring a directory for changes.

```
int pascal FS_FINDNOTIFYFIRST (pcdfsi, pcdfsd, pName, iCurDirEnd,
                               attr, pHandle,
                               pData, cbData, pcMatch,
                               level, timeout)

struct cdfsi *   pcdfsi;
struct cdfs *   pcdfsd;
char *          pName;
unsigned long    iCurDirEnd;
unsigned long    attr;
unsigned long *  pHandle;
char            pData;
unsigned long    cbData;
unsigned long *  pcMatch;
unsigned long    level;
unsigned long    timeout;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfs

pointer to file-system dependent working directory structure.

pName

pointer to asciiz name of file/directory. Wildcard characters are allowed only in the last component. The FSD does not need to verify this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

attr

bit field that governs the match. Any directory entry whose attribute bit mask is a subset of attr and whose name matches that in pName should be returned. See FS_FINDFIRST for explanation.

pHandle

pointer to directory handle. The FSD must allocate a handle for the directory monitoring continuation information and store it here. This handle will be passed to FS_FINDNOTIFYNEXT to continue directory monitoring. The FSD does not need to verify this pointer.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData

length of the application data area in bytes.

pcMatch

pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to verify this pointer.

level

information level to be returned. Level selects among a series of structures of data to be returned. See DOSFINDNOTIFYFIRST for information. The level passed to the FSD will be valid.

timeout

millisecond timeout. The FSD will wait until either the timeout has expired, the buffer is full, or the specified number of entries returned before returning to the caller.

FS_FINDNOTIFYNEXT - Resume reporting directory changes

Purpose

Resume reporting of directory or file changes.

```
int pascal FS_FINDNOTIFYNEXT (handle, pData, cbData, pcMatch)
unsigned long    handle;
char *          pData;
unsigned long    cbData;
unsigned long *  pcMatch;
unsigned long    timeout;
```

Where

handle

directory handle. This handle was returned by the FSD and is associated with a previous FS_FINDNOTIFYFIRST or FS_FINDNOTIFYNEXT call.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). The FSD will fill in this area with a set of packed, variable-length structures that contain the requested data and matching file names.

cbData

length of the application data area in bytes.

pcMatch

pointer to number of matching entries. The FSD will return at most this number of entries. The FSD will store into it the number of entries actually placed in the data area. The FSD does not need to verify this pointer.

timeout

millisecond timeout. The FSD will wait until either the timeout has expired, the buffer is full, or the specified number of entries returned before returning to the caller.

Remarks

pcMatch is the number of changes required to directories or files that match the **pName** target and **attr** specified during a related, previous **FS_FINDNOTIFYFIRST**. The file system uses this field to return the number of changes that actually occurred since the issue of the present **FS_FINDNOTIFYNEXT**.

FS_FLUSHBUF - Commit file buffers

Purpose

Flushes cache buffers for a specific volume.

```
int pascal FS_FLUSHBUF (hVPB, flag)
unsigned long  hVPB;
unsigned long  flag;
```

Where

hVPB

handle to volume for flush.

flag

indicator for discarding of cached data.

flag == 0 indicates cached data may be retained.

flag == 1 indicates the FSD will discard any cached data after flushing it to the specified volume.

All other values reserved.

FS_FSCTL - File System Control

Purpose

Allow an extended standard interface between an application and a file system driver.

```
int pascal FS_FSCTL (pArgdat, iArgType, func,
                    pParm, lenParm,
                    pData, lenData)

union argdat * pArgDat
unsigned long  iArgType;
unsigned long  func;
char *        pParm;
unsigned long  cbParm;
char *        pData;
unsigned long  cbData;
```

Where

pArgDat

pointer to union whose contents depend on iArgType. Union is:

```
union argdat {

    /* pArgType = 1, FileHandle directed case */
    struct sf {
        struct sffsi * psfsi;
        struct sffsd * psfsd;
    };

    /* pArgType = 2, Pathname directed case */
    struct cd {
        struct cdfsi * pcdfsi;
        struct cdbsd * pcdbsd;
        char *        pPath;
        unsigned long  iCurDirEnd;
    };
};
```

```

};

/* pArgType = 3, FSD Name directed case */
/* garbage */
};

```

iArgType

indicator of argument type.

- iArgType = 1 means that pArgDat->sf.psfsi and pArgDat->sf.psfsd point to an sfsi and sfsd, respectively.
- iArgType = 2 means that pArgDat->cd.pcdfsi and pArgDat->cd.pcdfsd point to a cdfsi and cdfsd, pArgDat->cd.pPath points to a canonical pathname, and pArgDat->cd.iCurDirEnd gives the index of the end of the current directory in pPath. The FSD does not need to verify the pPath pointer.
- iArgType = 3 means that the call was FSD name routed, and pArgDat is a NULL pointer.

func

indicator of function to perform.

func == 1 indicates request for new error code information.

pParm

address of application input parameter area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

lenParm

byte length of application input parameter area.

pData

address of application output data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

lenData

byte length of application output data area.

Remarks

The accessibility of the parameter and data buffers has not been validated by the DOS. FS_PROBEBUF must be used.

All FSDs must support func == 1 to return new error code information.

FS_FSINFO - File System Information

Purpose

Returns/Sets information for a filesystem device.

```

int pascal FS_FSINFO (flag, hVPB, pData, cbData, level)
unsigned long    flag;
unsigned long    hVPB;
char *          pData;
unsigned long    cbData;
unsigned long    level;

```

Where

flag

indicates retrieval of information vs setting information.

flag == 0 indicates retrieving information.

flag == 1 indicates setting information on the media.

All other values reserved.

hVPB

handle to volume of interest.

pData

address of application output data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

cbData

length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR_BUFFER_OVERFLOW. For flag == 1, this is the length of data to be sent to the file system.

level

information level to be returned. Level selects among a series of structures of data to be returned or set. See DOSQFSINFO and DOSSETFSINFO for information.

Remarks

None.

FS_INIT - File system driver initialization

Purpose

Request file system driver initialization.

```
int pascal FS_INIT (szParm)
char * szParm;
```

Where

szParm

pointer to asciiz parameters following the config.sys IFS= command that loaded the FSD. The FSD does not need to verify this pointer.

Remarks

This call is made during system initialization to allow the FSD to perform actions necessary for beginning operation. The FSD may successfully initialize by returning 0 or may reject installation (invalid parameters, incompatible hardware, etc) by returning the appropriate error code. If rejection is selected, all FSD selectors and segments are released.

FS_IOCTL - I/O Control for Devices

Purpose

Perform control functions on the device specified by the opened device handle.

```
int pascal FS_IOCTL (psffsi, psffsd, cat, func,
                    pParm, lenParm,
                    pData, lenData)

struct sffsi * psffsi;
struct sffsd * psffsd;
unsigned long  cat;
unsigned long  func;
char *        pParm;
unsigned long  cbParm;
char *        pData;
unsigned long  cbData;
```

Where

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

cat

category of function to be performed.

func

function within category to be performed.

pParm

address of application input parameter area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). A null value indicates that the parameter is unspecified for this function.

lenParm

byte length of application input parameter area. If lenParm is 0 and pParm is not null, it means the data buffer length is unknown due to the request being submitted via an old IOCTL or DOSDEVIOCTL interface.

pData

address of application output data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). A null value indicates that the parameter is unspecified for this function.

lenData

byte length of application output data area. If lenData is 0 and pData is not null, it means the data buffer length is unknown due to the request being submitted via an old IOCTL or DOSDEVIOCTL interface.

FS_MKDIR - Make Subdirectory

Purpose

Create the specified directory.

```
int pascal FS_MKDIR (pcdfsi, pcdfsd,
                    pName, iCurDirEnd,
                    pEABuf)

struct cdfsi * pcdfsi;
struct cdfs * pcdfsd;
char * pName;
unsigned long iCurDirEnd;
char * pEABuf;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfs

pointer to file-system dependent working directory structure.

pName

pointer to asciiz name of directory to be created. The FSD does not need to verify this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

pEABuf

pointer to extended attribute buffer. This buffer contains attributes that will be set upon creation of the new directory. If NULL, no extended attributes are to be set. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

Remarks

A new directory called pName should be created if possible. The standard directory entries '.' and '..' should be put into the directory.

FS_MOUNT - Mount/Unmount volumes

Purpose

Examination of a volume by an FSD to see if it recognizes the file system format.

```
int pascal FS_MOUNT (flag, pvpfsi, pvpfsd, hVPB, pBoot)
unsigned long flag;
struct vpfsi * pvpfsi;
struct vpfsd * pvpfsd;
unsigned long hVPB;
char * pBoot;
```

Where

flag

indicates operation requested.

flag == 0 indicates that the FSD is requested to mount or accept a volume.

flag == 1 indicates that the FSD is being advised that the specified volume has been removed.

flag == 2 indicates that the FSD is requested to release all internal storage assigned to that volume as it has been removed from its drive and the last kernel-managed reference to that volume has been removed.

flag == 3 indicates that the FSD is requested to accept the volume regardless of recognition in preparation for formatting for use with the FSD.

All other values reserved. The value passed to the FSD will be valid.

vpfsi

pointer to file-system-independent portion of VPB. If the media contains an OS/2-recognizable boot sector, then the vpi_vid field contains the 32-bit identifier for that volume. If the media does not contain such a boot sector, the FSD must generate a unique label for the media and place it into the vpi_vid field.

vpfsd

pointer to file-system-dependent portion of VPB. The FSD may store information as necessary into this area.

hVPB

handle to volume.

pBoot

pointer to sector 0 read from the media. This pointer is ONLY valid when flag == 0. The buffer the pointer refers to MUST NOT BE MODIFIED. The pointer is always valid and does not need to be verified when flag == 0; if a read error occurred, the buffer will contain zeroes.

Remarks

The FSD should examine the volume presented and determine whether it recognizes the file system. If so, it should return zero after having filled in appropriate parts of vpfsi and vpfsd. The vpi_vid and vpi_text fields must be filled in by the FSD. If the FSD has an OS/2 format boot sector, it must convert the label from the media into ascii form. The vpi_hDev field will be filled in by OS/2. If the volume is unrecognized, the driver should return non-zero.

The vpi_text and vpi_vid must be updated by the FSD each time these values change.

The contents of the vpfsd are as follows:

(FLAG = 0)

The FSD is expected to issue an FSD_FINDDUPHVPB to see if a duplicate VPB exists if one does exist the VPB fs dependent area of the new VPB is invalid and the new VPB will be unmounted after the FSD returns from the MOUNT. The FSD is expected to update the fs dependent area of the old duplicate VPB.

If no duplicate VPB exists the FSD should initialize the fs dependent.

(FLAG = 1)

VPB fs dependent part is same as when FSD last modified it.

(FLAG = 2)

VPB fs dependent part is same as when FSD last modified it.

After media recognition time, the volume parameters may be examined using the **FSH_GETVOLPARM** call. The volume parameters should not be changed after media recognition time.

During a mount request, the FSD may examine other sectors on the media by using **FSH_DOVOLIO** to perform the I/O. If an uncertain-media return is detected, the FSD is expected to clean up and return **ERROR_UNCERTAIN_MEDIA** in order to allow the volume mount logic to restart on the newly-inserted media. The FSD must provide the buffer to use for additional I/O.

The kernel manages the VPB via a ref count. All volume-specific objects are labelled with the appropriate volume handle and represent references to the VPB. When all kernel references to a volume disappear, **FS_MOUNT** is called with flag == 2, indicating a dismount request.

When the kernel detects that a volume has been removed from its drive, but there are still outstanding references to the volume, **FS_MOUNT** is called with flag == 1 to allow the FSD to drop clean (or other regenerable) data for the volume. Data which is dirty and cannot be regenerated should be kept so that it may be written to the volume when it is remounted in the drive.

When a volume is to be formatted for use with an FSD, the kernel calls the FSD's **FS_MOUNT** entry with flag == 3 to allow the FSD to prepare for the format operation. The FSD should accept the volume even if it is not a volume of the type that FSD recognizes, since the point of format is to change the filesystem on the volume. The operation may be failed if formatting doesn't make any sense. (For example, an FSD which supports only CD-ROM.)

Since the hardware does not allow for kernel-mediated removal of media, it is certain that the unmount request is issued when the volume is not present in any drive.

FS_MOVE - Move a File or Subdirectory

Purpose

Moves (renames) the specified file or subdirectory.

```
int pascal FS_MOVE (pcdfsi, pcdfsd,
                    pSrc, iSrcCurDirEnd,
                    pDst, iDstCurDirEnd)

struct cdfsi * pcdfsi;
struct cdfsd * pcdfsd;
char * pSrc;
unsigned long iSrcCurDirEnd;
char * pDst;
unsigned long iDstCurDirEnd;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfsd

pointer to file-system dependent working directory structure.

pSrc

pointer to asciiz name of source file/directory. The FSD does not need to verify this pointer.

iSrcCurDirEnd

index of the end of the current directory in pSrc. This is used to optimize FSD path processing. If iSrcCurDirEnd == -1 there is no current directory relevant to the source name text.

pDst

pointer to asciiz name of destination file/directory. The FSD does not need to verify this pointer.

iDstCurDirEnd

index of the end of the current directory in pDst. This is used to optimize FSD path processing. If iDstCurDirEnd == -1 there is no current directory relevant to the destination name text.

Remarks

The file specified in filename should be moved or renamed to be destfile if possible.

Neither the source nor the destination filename may contain wildcard characters.

FS_NEWSIZE - Change File's Logical Size

Purpose

Changes a file's logical (EOD) size.

```
int pascal FS_NEWSIZE (psffsi, psffsd, len)
struct sffsi * psffsi;
struct sffsd * psffsd;
unsigned long len;
```

Where

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

len

desired new length of file.

Remarks

The file system driver should attempt to set the size (EOD) of the file to newsize and update sfi_size if successful. If the new size is larger than the currently allocated size, the file system driver should to the extent possible arrange for efficient access to the newly allocated storage.

FS_OPENCREATE - Open a File

Purpose

Opens (or creates) the specified file.

```
int pascal FS_OPENCREATE (pcdfsi, pcdfsd,
                           pName, iCurDirEnd,
                           psffsi, psffsd,
                           fhandflag, openflag,
```

```

                                pAction, attr,
                                pEABuf)

struct cdfsi *    pcdfsi;
struct cdfsd *    pcdfsd;
char *           pName;
unsigned long     iCurDirEnd;
struct sffsi *    psffsi;
struct sffsd *    psffsd;
unsigned long     fhandflag;
unsigned long     openflag;
unsigned long *   pAction;
unsigned long     attr;
char *           pEABuf;

```

Where

pcdfsi

pointer to file-system independent working directory structure. The contents of this structure are invalid for direct access opens.

pcdfsd

pointer to file-system dependent working directory structure. The contents of this structure are invalid for direct access opens.

pName

pointer to asciiz name of file to be opened. The FSD does not need to verify this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device. This value is invalid for direct access opens.

psffsi

pointer to file-system independent portion of open file instance.

psffsd

pointer to file-system dependent portion of open file instance.

fhandflag

indicates the desired sharing mode and access mode for the file handle. See the API documentation for the OpenMode parameter for DOSOPEN. An additional access mode 3 is defined when the file is being opened on behalf of the OS/2 loaded for purposes of executing a file or loading a module. If the file system does not support an executable attribute, it should treat this access mode as open for reading. The value of fhandflag passed to the FSD will be valid.

openflag

indicates the action taken when the file is present or absent. See the API documentation for the OpenFlag for DOSOPEN. The value of openflag passed to the FSD will be valid. This value is invalid for direct access opens.

pAction

location where FSD returns a description of the action taken as governed by openflag. The FSD does not need to verify this pointer. The contents of Action are invalid on return for direct access opens.

attr

OS/2 file attributes. This value is invalid for direct access opens.

pEABuf

pointer to extended attribute buffer. This buffer contains attributes that will be set upon creation of a new file or upon replacement of an existing file. If NULL, no extended attributes are to be set. Addressing of this data area has not been

validated by the kernel (See FSH_PROBEBUF). The contents of EABuf are invalid on return for direct access opens.

Remarks

The sharing mode may be zero if this is a request to open a file from the 3.x box in compatibility mode or for an FCB request.

FCB requests for read-write access to a read-only file should be mapped to read-only access and reflected in the sfi_mode field by the FSD. An FCB request is indicated by the high bit set in the sfi_mode field.

Also on entry, the sfi_hvpb field is filled in. If the file's logical size (EOD) is specified, it will be passed in the sfi_size field. To the extent possible, the file system should try to allocate this much storage for efficient access.

Extended attributes are set for 1) the creation of a new file, 2) the truncation of an existing file, and 3) the replacement of an existing file. They are not set for a normal open.

If the standard OS/2 file creation attributes have been specified, they will be passed in the attr field. To the extent possible, the file system should interpret the extended attributes and apply them to the newly created or existing file. Extended attributes (EAs) that the file system does not itself use should be retained with the file and not discarded or rejected.

FSDs are required to support direct access opens. These are indicated by a bit set in the sffsi.sfi_mode field. See DOSOPEN for information. Some of the parameters passed to the FSD for direct access opens are invalid, as described above.

On a successful return, the following fields in the sffsi structure must be filled in by the file system driver: sfi_size and all the time and date fields.

The file-system dependent portion of open file instance passed to the FSD for FS_OPENCREATE will always be uninitialized.

Infinite FCB opens of the same file by the same 3xbox process is supported. The first open is passed through to the FSD. Subsequent opens are not seen by the FSD.

FS_PATHINFO - Query/Set a File's Information

Purpose

Returns information for a specific path or file.

```
int pascal FS_PATHINFO (flag, pcdfsi, pcdfsd,
                        pName, iCurDirEnd,
                        level, pData, cbData)
```

```
unsigned long  flag;
struct cdfsi * pcdfsi;
struct cdfsd * pcdfsd;
char *        pName;
unsigned long  iCurDirEnd;
unsigned long  level;
char *        pData;
unsigned long  cbData;
```

Where

flag

indicates retrieval of information vs setting information.

flag == 0 indicates retrieving information.

flag == 1 indicates setting information on the media.

All other values reserved.

<u>pcdfsi</u>	pointer to file-system independent working directory structure.
<u>pcdfsd</u>	pointer to file-system dependent working directory structure.
<u>pName</u>	pointer to asciiz name of file or directory for which information is to be retrieved or set. The FSD does not need to verify this pointer.
<u>iCurDirEnd</u>	index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.
<u>level</u>	information level to be returned. Level selects among a series of structures of data to be returned or set.
<u>pData</u>	address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF). When retrieval (flag == 0) is specified, the FSD will place the information into the buffer. When outputting information to a file (flag == 1), the FSD will retrieve that data from the application buffer.
<u>cbData</u>	length of the application data area. For flag == 0, this is the length of the data the application wishes to retrieve. If there is not enough room for the entire level of data to be returned, the FSD will return ERROR_BUFFER_OVERFLOW. For flag == 1, this is the length of data to be applied to the file.

Remarks

See DOSQPATHINFO and DOSSETPATHINFO for information level descriptions.

The FSD will not be called for DOSQPATHINFO level 5.

FS_PROCESSNAME - Allow FSD to modify name after OS/2 canor

Purpose

Allow FSD to modify filename to its own specification after the OS/2 canonicalization process has completed.

```
int pascal FS_PROCESSNAME (pNameBuf)
char * pNameBuf;
```

Where

<u>pNameBuf</u>	pointer to ASCIIZ pathname. The FSD should modify the pathname in place. The buffer is guaranteed to be the length of the maximum path. The FSD does not need to verify this pointer.
-----------------	---

Remarks

The resulting name must be within the maximum path length returned by DOSQSYSINFO.

This routine allows the FSD to enforce a different naming convention than OS/2. For example, an FSD could remove blanks embedded in component names or return an error if it found such blanks. It is called after the OS/2 canonicalization process has succeeded. It is not called for FSH_CANONICALIZE.

FS_READ - Read from a File

Purpose

Read the specified number of bytes from a file to a buffer location.

```
int pascal FS_READ (psffsi, psffsd, pData, pLen)
struct sffsi *      psffsi;
struct sffsd *      psffsd;
char *              pData;
unsigned long *      pLen;
```

Where

psffsi

pointer to file-system independent portion of open file instance. sfi_position is the location within the file where the data is to be read from. The FSD should update the sfi_position field.

psffsd

pointer to file-system dependent portion of open file instance.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

pLen

pointer to length of the application data area. On input, this is the number of bytes that to be read. On output, this is the number of byte successfully read. If the application data area is smaller than the length, no transfer is to take place. The FSD will not be called for zero length reads. The FSD does not need to verify this pointer.

FS_RMDIR - Remove Subdirectory

Purpose

Removes a subdirectory from the specified disk.

```
int pascal FS_RMDIR (pcdfsi, pcdfsd,
                    pName, iCurDirEnd)
struct cdfsi *      pcdfsi;
struct cdfsd *      pcdfsd;
char *              pName;
unsigned long        iCurDirEnd;
```

Where

pcdfsi

pointer to file-system independent working directory structure.

pcdfsd

pointer to file-system dependent working directory structure.

pName

pointer to asciiz name of directory to be removed. The FSD does not need to verify this pointer.

iCurDirEnd

index of the end of the current directory in pName. This is used to optimize FSD path processing. If iCurDirEnd == -1 there is no current directory relevant to the name text, i.e. a device.

Remarks

OS/2 has already assured that the directory being removed is not the current directory nor the parent of any current directory of any process.

The FSD should not remove any directory that has entries other than . and .. in it.

FS_SETSWAP - Notification of swap-file ownership

Purpose

Perform whatever actions are necessary to support the swapper.

```
int pascal FS_SETSWAP (psffsi, psffsd)
struct sffsi * psffsi;
struct sffsd * psffsd;
```

Where

psffsi

pointer to file-system independent portion of open file instance of the swapper file.

psffsd

pointer to file-system dependent portion of open file instance.

Note

Swapping will not begin until this calls returns successfully. This call will be made during system initialization.

The FSD will make all segments that are relevant to swap-file I/O non-swappable (see FSH_FORCENOSWAP). This includes any data and code segments accessed during a read or write.

Any FSD that manages writeable media may be the swapper file system.

FS_SETSWAP may be called more than once for the same or different volumes or FSDs.

FS_WRITE - Write to a File

Purpose

Write the specified number of bytes to a file from a buffer location.

```
int pascal FS_WRITE (psffsi, psffsd,
                    pData, pLen)
struct sffsi *      psffsi;
struct sffsd *      psffsd;
char *              pData;
unsigned long *      pLen;
```

Where

psffsi

pointer to file-system independent portion of open file instance. sfi_position is the location within the file where the data is to be written to. The FSD should update the sfi_position and sfi_size fields.

psffsd

pointer to file-system dependent portion of open file instance.

pData

address of application data area. Addressing of this data area has not been validated by the kernel (See FSH_PROBEBUF).

pLen

pointer to length of the application data area. On input, this is the number of bytes that to be written. On output, this is the number of byte successfully written. If the application data area is smaller than the length, no transfer is to take place. A zero length write indicates truncation. The FSD does not need to verify this pointer.

FS Services Supplied By OS/2

An FSD may access OS/2 services via two different paths: via DOS API calls at initialization time, via dynlinks to OS/2 FS-help services at task time.

The set of DOS API calls available at initialization time is equivalent to those available to device drivers at initialization time. (See IPL section near front of spec.) This includes DosOpen, DosRead, DosWrite, DosDevIoCtl and DosClose among others.

FS Help Routines

FSDs are loaded as dynlink libraries and are able to import services provided by the kernel. These services can be called directly by the file system, passing the relevant parameters.

No validation of input parameters is done unless otherwise specified. The FSD should call FSH_PROBEBUF where appropriate before calling the FS help routine.

When any service returns an error code, the FSD must wind out of the particular FS call as soon as possible and return the specific error code

from the helper to the FS router.

There are many deadlocks that may occur as a result of operations issued by FSDs. OS/2 provide no means whereby deadlocks between file systems and applications can be detected.

The FSD helper routines are:

<u>FSH_ADDSHARE</u>	Add a name to the sharing set
<u>FSH_BUFSTATE</u>	Get or set buffer state
<u>FSH_CANONICALIZE</u>	Convert pathname to canonical form
<u>FSH_CRITERORR</u>	Signal a hard error to the daemon
<u>FSH_DEVIOCTL</u>	Send IOCTL request to device driver
<u>FSH_DOVOLIO</u>	Volume-based sector-oriented transfer
<u>FSH_DOVOLIO2</u>	Send volume-based IOCTL request to device driver
<u>FSH_FINDCHAR</u>	Find first occurrence of char in string
<u>FSH_FINDDUPHVPB</u>	Locates equivalent hVPBs
<u>FSH_FLUSHBUF</u>	Flush buffered data to a volume
<u>FSH_FORCENOSWAP</u>	Force segments permanently into memory
<u>FSH_GETBUF</u>	Buffered sector read
<u>FSH_GETFIRSTOVERLAPBUF</u>	Locates first buffer overlapping range
<u>FSH_GETVOLPARM</u>	Get VPB data from VPB handle
<u>FSH_INTERR</u>	Signal an internal error
<u>FSH_ISCURDIRPREFIX</u>	Test for a prefix of a current directory
<u>FSH_LOADCHAR</u>	Load character from a string
<u>FSH_LOCKRANGE</u>	Lock or unlock a range of memory
<u>FSH_PREVCHAR</u>	Move backward in string
<u>FSH_PROBEBUF</u>	User address validity check
<u>FSH_QSYSINFO</u>	Query system information
<u>FSH_RELEASEBUF</u>	Release the owned buffer
<u>FSH_REMOVESHARE</u>	Remove a name from the sharing set

<u>FSH_SEMCLEAR</u>	Request a semaphore
<u>FSH_SEMREQUEST</u>	Request a semaphore
<u>FSH_SEMSET</u>	Set a semaphore
<u>FSH_SEMSETWAIT</u>	Set a semaphore and wait for clear
<u>FSH_SEMWAIT</u>	Wait for clear
<u>FSH_STORECHAR</u>	Store character into string
<u>FSH_WILDMATCH</u>	Match using OS/2 wildcards
<u>FSH_YIELD</u>	Yield CPU to higher priority threads

FSH_ADDSHARE - Add a name to the share set

FSH_ADDSHARE adds a name to the currently active sharing set. FSDs will use this call when executing deletes or renames with wildcard characters. FSH_ADDSHARE returns a share handle that must be used with a succeeding FSH_REMOVESHARE.

```
int pascal FSH_ADDSHARE (pName, mode, hVPB, phShare)
char *      pName;
unsigned long mode;
unsigned long hVPB;
unsigned long * phShare;
```

Where

pName

pointer to asciiz name to be added into the share set. The name must be in canonical form: no '.' or '..' components, uppercase, no metacharacters, and full pathname specified.

mode

sharing mode and access mode as defined in the DOSOPEN API. All other bits (Direct Open, Write-Through, etc) must be zero.

hVPB

handle to volume where the named object is presumed to exist.

phShare

pointer to location where a share handle is stored. This handle may be passed to FSH_REMOVESHARE.

Returns

Error code if error detected, 0 otherwise.

ERROR_SHARING_VIOLATION - the file is open with a conflicting sharing/access mode.

ERROR_TOO_MANY_OPEN_FILES - there are too many files open at the present time.

ERROR_SHARING_BUFFER_EXCEEDED - there is not enough memory to hold sharing information.

ERROR_INVALID_PARAMETER - invalid bits in mode.

ERROR_FILENAME_EXCED_RANGE - pathname is too long.

Note

Do not call FSH_ADDSHARE for character devices.

FSH_ADDSHARE may block.

To help avoid deadlocks, FSH_ADDSHARE should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_BUFSTATE - Get or set buffer state

FSH_BUFSTATE is used retrieve or set the state of a buffer.

```
int pascal FSH_BUFSTATE (pBuf, flag, pState)
char *      pBuf;
unsigned long flag;
unsigned long * pState;
```

Where

pBuf

pointer to buffer. This pointer was returned to the FSD by FSH_GETBUFFER

flag

indicates get or set buf state.

flag == 0 indicates retrieve state

flag == 1 indicates set state

pState

for set, points to new state of buffer on input. For retrieve, points to state of buffer.

State == 0x00000080 indicates dirty buffer.

All other values are reserved.

Returns

Error code if error detected, 0 otherwise.

ERROR_INVALID_PARAMETER - pointer to buffer is invalid or reserved state bits are set.

Note

To set the opposite of a defined state, call FSH_BUFSTATE with the bit not set. For example, to set a buffer not dirty, pass 0x00000000 as the new state.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_CANONICALIZE - Convert pathname to canonical form

FSH_CANONICALIZE is used to convert a pathname to a canonical form by processing '.'s and '..'s, uppercasing, and prepending the current directory to non-absolute paths.

```
int pascal FSH_CANONICALIZE (pPathName, cbPathBuf, pPathBuf)
char *      pPathName;
unsigned long cbPathBuf;
char *      pPathBuf;
```

Where

<u>pPathName</u>	pointer to asciiz pathname to be canonicalized.
<u>cbPathBuf</u>	length of pathname buffer.
<u>pPathBuf</u>	pointer to buffer to copy canonicalized path into.

Returns

Error code if error detected, 0 otherwise.

ERROR_PATH_NOT_FOUND - invalid pathname - too many '..'s.

ERROR_BUFFER_OVERFLOW - pathname is too long.

Note

This routine process DBCS characters properly.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_CRITERROR - Signal a hard error to the daemon

FSH_CRITERROR is used to pass an error from the current thread to the hard error daemon, block waiting for a response, and return the response to the caller.

```
int pascal FSH_CRITERROR (cbMessage, pMessage,
                          nSubs, pSubs, fAllowed)
unsigned long cbMessage;
char *      pMessage;
unsigned long nSubs;
char *      pSubs;
unsigned long fAllowed;
```

Where

<u>cbMessage</u>	length of message template
<u>pMessage</u>	pointer to message template. This may contain replaceable paramters in the format used by the message retriever.
<u>nSubs</u>	number of replaceable parameters.
<u>pSubs</u>	pointer to replacement text. The replacement text is a packed set of ASCIIZ strings.
<u>fAllowed</u>	bit mask of allowed actions: Bit 0x00000001 on indicates FAIL allowed Bit 0x00000002 on indicates ABORT allowed Bit 0x00000004 on indicates RETRY allowed Bit 0x00000008 on indicates IGNORE allowed All other bits are reserved and must be zero.

Returns

Action to be taken:
0x00000000 ignore
0x00000001 retry
0x00000003 fail

Note

If the user responds with an action that is not allowed, it is treated as FAIL. If FAIL is not allowed, it is treated as ABORT. ABORT is always allowed.

When ABORT is the final action, OS/2 does not return this as an indicator, only that FAIL was returned. The actual ABORT operation is generated when this thread of execution is about to return to user code,

The maximum length of the template is 128 bytes (including the NUL). The maximum length of the message with text substitutions is 512 bytes. The maximum number of substitutions is 9.

FSH_CRITERROR may block.

To help avoid deadlocks, FSH_CRITERROR should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_DEVIOCTL - Send IOCTL request to device driver

FSDs call FSH_DEVIOCTL to control device driver operation independently from I/O operations. This is typically in filtering DOSDEVIOCTL requests when passing the request on to the device driver.

```

int pascal FSH_DEVIOCTL (hDev, sfn, cat, func,
                        pParm, cbParm, pData, cbData)
unsigned long hDev;
unsigned long sfn;
unsigned long cat;
unsigned long func;
char * pParm;
unsigned long cbParm;
char * pData;
unsigned long cbData;

```

Where

hDev

device handle obtained from VPB

sfn

system file number from open instance that caused the FSH_DEVIOCTL call. This field should be passed unchanged from the sfi_selfsfn field. If no open instance corresponds to this call, this field should be set to 0xFFFFFFFF.

cat

category of IOCTL to perform

func

function within category of IOCTL

pParm

long address to parameter area

cbParm

length of parameter area

pData

long address to data area

cbData

length of data area

Returns

Error code if error detected, 0 otherwise.

ERROR_INVALID_FUNCTION - the function supplied is incompatible with the category supplied and the device handle supplied.

ERROR_INVALID_CATEGORY - the category supplied is incompatible with the function supplied and the device handle supplied.

Device driver error code

Note

FSH_DEVIOCTL may block.

To help avoid deadlocks, FSH_DEVIOCTL should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_DOVOLIO - Volume-based sector-oriented transfer

FSH_DOVOLIO performs I/O to the specified volume. It formats a device driver request packet for the requested I/O, locks the data transfer region, calls the device driver, and reports any errors to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DOSError are done within the call to FSH_DOVOLIO.

```
int pascal FSH_DOVOLIO (operation, hVPB, pData, pcSec, iSec)
unsigned long    operation;
unsigned long    hVPB;
char *          pData;
unsigned long *  pcSec;
unsigned long    iSec;
```

Where

operation

bit mask indicating read/read-bypass/write/write-bypass and verify-after-write operation to be performed.

Bit 0x00000001 off indicates read.

Bit 0x00000001 on indicates write.

Bit 0x00000002 off indicates no bypass.

Bit 0x00000002 on indicates cache bypass.

Bit 0x00000004 off indicates no verify-after-write operation.

Bit 0x00000004 on indicates verify-after-write.

Bit 0x00000008 off indicates errors signalled to the hard error daemon.

Bit 0x00000008 on indicates hard errors will be returned directly.

All other bits are reserved and must be zero.

hVPB

volume handle for source of I/O

pData

long address of user transfer area

pcSec

pointer to number of sectors to be transferred. On return this is the number of sectors successfully transferred.

iSec

sector number of first sector of transfer

Returns

Error code if operation failed, 0 otherwise.

ERROR_PROTECTION_VIOLATION - the supplied address/length is not valid.

ERROR_UNCERTAIN_MEDIA - the device driver can no longer reliably tell if the media has been changed. This occurs only within the context of an FS_MOUNT call.

ERROR_TRANSFER_TOO_LONG - transfer is too long for device

Device-driver/device-manager [errors listed](#).

Note

FSH_DOVOLIO may be used at all times within the FSD. When called within the scope of a FS_MOUNT call, it applies to the volume in the drive without regard to which volume it may be. However, since volume recognition is NOT complete until the FSD returns to the FS_MOUNT call, the FSD must take care when an ERROR_UNCERTAIN_MEDIA is returned. This indicates that the media has gone uncertain while we are trying to identify the media in the drive. This may indicate that the volume that the FSD was trying to recognize was removed. In this case, the FSD must release any resources attached to the hVPB passed in the FS_MOUNT call and return ERROR_UNCERTAIN_MEDIA to the FS_MOUNT call. This will direct the volume tracking logic to restart the mount process.

If an error occurs during the transfer and an OS/2 buffer is owned, it will be released before signalling the hard error daemon.

Verify-after-write specified on a read is ignored.

FSH_DOVOLIO may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_DOVOLIO2 - Send volume-based IOCTL request to device dri

FSDs call FSH_DOVOLIO2 to control device driver operation independently from I/O operations. This routine supports volume management for IOCTL operations. Any errors are reported to the hard error daemon before returning to the FSD. Any retries indicated by the hard error daemon or actions indicated by DOSERROR are done within the call to FSH_DOVOLIO2.

```
int pascal FSH_DOVOLIO2 (hDev, sfn, cat, func, pParm,
                        cbParm, pData, cbData, flag)
unsigned long  hDev;
unsigned long  sfn;
unsigned long  cat;
unsigned long  func;
char *        pParm;
unsigned long  cbParm;
char *        pData;
unsigned long  cbData;
unsigned long  flag;
```

Where

hDev

device handle obtained from VPB

sfn

system file number from open instance that caused the FSH_DEVIOCTL call. This field should be passed unchanged from the sfi_selfsfn field. If no open instance corresponds to this call, this field should be set to 0xFFFFFFFF.

cat

category of IOCTL to perform

func

function within category of IOCTL

pParm

long address to parameter area

cbParm

length of parameter area

pData

long address to data area

cbData

length of data area

flag

bit mask indicating error behavior

Bit 0x00000008 off indicates errors signalled to the hard error daemon.

Bit 0x00000008 on indicates hard errors will be returned directly.

All other bits are reserved and must be zero.

Returns

Error code if error detected, 0 otherwise.

ERROR_INVALID_FUNCTION - the function supplied is incompatible with the category supplied and the device handle supplied.

ERROR_INVALID_CATEGORY - the category supplied is incompatible with the function supplied and the device handle supplied.

Device driver error code

Note

The purpose of this routine is to enable volume tracking with ioctl's. It is not available at the API level.

FSH_DOVOLIO2 may block.

To help avoid deadlocks, FSH_DOVOLIO2 should not be called while owning an OS/2 buffer.

System does normal volume checking for this request.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_FINDCHAR - Find first occurrence of char in string

Provides mechanism for find the first occurrence of any one of a set of characters in an ASCIIZ string taking into account DBCS considerations.

```
int pascal FSH_FINDCHAR (nChars, pChars, ppStr)
unsigned long nChars;
char *        pChars;
char * *      ppStr;
```

Where

nChars

number of chars in search list.

pChars

array of chars to search for. These cannot be DBCS characters. The NULL

character cannot be searched for.

ppStr

pointer to character pointer to begin search from. This pointer is updated upon return to point to the character found. This must be an ASCII string.

Returns

Error code if match failed, 0 otherwise.

ERROR_CHAR_NOT_FOUND - none of the characters were found.

Note

The search will continue until a matching character is found or the end of the string is found.

The FSD is responsible for verifying the string pointers and checking for segment boundaries.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_FINDDUPHVPB - Locate equivalent hVPB

Provides mechanism for identifying previous instance of a volume during the FS_MOUNT process. When OS/2 is recognizing a volume, it calls the FSD to mount the volume. At this point, the FSD may elect to allocate storage and buffer data for that volume. The mount process will allocate a new VPB whenever the media becomes uncertain (the device driver recognizes that it can no longer be certain that the media is unchanged). This VPB cannot be collapsed with a previously allocated VPB (due to a reinsertion of media) until the FS_MOUNT call returns. However, the previous VPB may have some cached data that must be updated from the media (the media may have been written while it was removed). FSH_FINDDUPHVPB allows the FSD to find this previous occurrence of the volume in order to update the cached information for the old VPB. Note that the newly created VPB will be unmounted if there is another, older VPB for that volume.

```
int pascal FSH_FINDDUPHVPB (hVPB, phVPB)
unsigned long hVPB;
unsigned long * phVPB;
```

Where

hVPB

handle to the volume to be found

phVPB

pointer to where handle of matching volume will be stored.

Returns

Error code if no matching VPB found. 0 otherwise.

ERROR_NO_ITEMS - there is no matching hVPB.

Note

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_FLUSHBUF - Flush buffered data to a volume

Flush buffer takes dirty sectors contained in the buffer cache on a particular media and writes them out. Optionally, the data can be discarded afterwards.

```
int pascal FSH_FLUSHBUF (hVPB, fDiscard)
unsigned long hVPB;
unsigned long fDiscard;
```

Where

hVPB

handle to the volume to be flushed

fDiscard

indicates disposition of cached data.

fDiscard == 0 indicates don't discard any buffers.

fDiscard == 1 indicates discard clean buffers

All other values are reserved.

Returns

Error code if any write failed. 0 otherwise.

ERROR_INVALID_PARAMETER - the value of Operation is invalid.

Device-driver/device-manager [errors listed](#).

Note

If fDiscard = 1 and a write error occurred, the data in the buffer(s) that generated the error is not discarded.

See note under FSH_GETBUF for interactions with other buffer calls.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_FLUSHBUF may block.

FSH_FORCENOSWAP - Force memory range permanently into me

An FSD may call FSH_FORCENOSWAP to force segments to be loaded into memory and marked non-swappable. All segments both in the load image of the FSD and those allocated via FSH_SEGALLOC are eligible to be marked. There is no way to undo the effect of FSH_FORCENOSWAP.

If an FSD is notified that it is managing the swapping media, it should make this call for the necessary segments.

```
int pascal FSH_FORCENOSWAP (pMem, cbMem)
char *pMem;
unsigned long cbMem;
```

Where

pMem

points to beginning of range of memory to make non-swappable.

cbMem

is number of bytes to make non-swappable.

Returns

Error code if error detected, 0 otherwise.

ERROR_NOT_ENOUGH_MEMORY - not enough physical memory to make memory non-swappable.

ERROR_SWAP_TABLE_FULL, ERROR_SWAP_FILE_FULL, ERROR_PMM_INSUFFICIENT_MEMORY - attempt to grow the swap file failed.

Note

FSH_FORCENOSWAP may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_GETFIRSTOVERLAPBUF - Locates buffer overlapping range

Provides mechanism for handling buffers that overlap a large transfer region within a file. When performing large transfers, the FSD needs to identify all buffers that contain data that overlap a particular region of the media. When writing to the media, these overlapping buffers must be filled in with the new data as they contain old data while when reading these buffers need to be copied over the data transferred from the media.

Typically, when a large transfer is requested, the FSD will use an algorithm along the following lines:

```
isecFirst = lowest sector number to be transferred
isecLast = highest sector number to be transferred

/* While there are sectors still to be transferred
 */
while (isecFirst <= sSecLast) {

    /* Find first buffer in cache that's in the transfer
     * range.
     */
    FSH_GETFIRSTOVERLAPBUF (hVPB, isecFirst, isecLast, &isecBuf,
                           &pbuf);

    /* Perform the I/O to the first unbuffered part
     */
    csec = isecBuf - isecFirst;
    if (csec != 0)
        FSH_DOVOLIO (... , : csec, isecFirst);

    /* Handle the buffered data appropriately
     */
    ...

    /* Release the buffer
     */
    FSH_RELEASEBUF ();

    /* Reduce range of data-to-be-transferred
     */
    isecFirst = isecBuf + 1;

    /* Continue until no more sectors
     */
}
```

```
int pascal FSH_GETFIRSTOVERLAPBUF (hVPB, isecFirst, isecLast,
                                   pisecBuf, ppBuf)
```

```

unsigned long hVPB;
unsigned long isecFirst;
unsigned long isecLast;
unsigned long * pisecBuf;
char * *      ppBuf;

```

Where

hVPB

handle for volume of I/O

isecFirst

logical sector number of beginning of range

isecLast

last logical sector number of range

pisecBuf

pointer to returned logical sector number of buffered sector

ppBuf

pointer to location where pointer to buffer data is returned

Returns

Error code if no overlapping buffer found. 0 otherwise.

ERROR_NO_ITEMS - no overlapping buffer was found.

Note

FSH_GETFIRSTOVERLAPBUF may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

See note under FSH_GETBUF for interactions with other buffer calls.

FSH_GETBUF - Buffered sector read

FSH_GETBUF is used to access the OS/2 sector cache to retrieve a sector from a particular volume. The pointer to the OS/2 buffer cache element is returned. The data in the buffer may be pre-read if desired.

```

int pascal FSH_GETBUF (iSec, fPreRead, hVPB, ppBuf)
unsigned long   iSec;
unsigned long   fPreRead;
unsigned long   hVPB;
char * *        ppBuf;

```

Where

iSec

sector number on the volume to return

fPreRead

indicates whether the sector should be pre-read.

fPreread == 0x00000000 indicates preread sector.

fPreread == 0x00000001 indicates no prereading of sector.

All other values are reserved.

hVPB

handle to the volume

ppBuf

pointer to location where pointer to buffer data is returned

Returns

Error code if operation failed, 0 otherwise.

ERROR_GETBUF_FAILED - the write to clear out a buffer failed or the read to fill the buffer in failed.

Note

FSH_GETFIRSTOVERLAPBUF, FSH_GETBUF and FSH_RELEASEBUF are used to obtain and release buffers. Any buffer that is owned by a thread is unavailable to all other threads until it is freed. At most one buffer may be owned by any thread.

Freeing occurs by calling FSH_FLUSHBUF, FSH_RELEASEBUF, or FSH_GETBUF to retrieve another sector.

The buffer returned is marked as being owned by the calling thread. If any OS/2 buffer is still owned by an FSD when the system call completes, the system will halt with an Internal Error, thus the FSD is responsible for ensuring that all system owned buffers are released before it returns to the router.

Not being preread is a performance optimization for when a file is being grown and a partial sector is being filled in.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_GETBUF may block.

FSH_GETVOLPARAM - Get VPB data from VPB handle

FSH_GETVOLPARAM allows an FSD to retrieve file-system-independent and -dependent data from a VPB. Since the FS router passes in a VPB handle individual FSDs need to map the handle into pointers to the relevant portions.

```
void pascal FSH_GETVOLPARAM (hVPB, ppVPBfsi, ppVPBfsd)
unsigned long hVPB;
struct vpfsi * * ppVPBfsi;
struct vpfds * * ppVPBfsd;
```

Where

hVPB

volume handle of interest

ppVPBfsi

location of where pointer to file-system- independent data is stored

ppVPBfsd

location of where pointer to file-system- dependent data is stored

Returns

Nothing

Note

FSH_GETVOLPARM will not block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_INTERR - Signal an internal error

For reliability, if an FSD detects an internal inconsistency during normal operation, the safest thing is for the FSD to shut down the system as a whole, the reason being that it is not clear if the system as a whole is in a state that allows normal execution to continue.

When an FSD calls FSH_INTERR, the address of the caller and the supplied message is displayed on the console. The system then halts.

```
void pascal FSH_INTERR (pMsg, cbMsg)
char *          pMsg;
unsigned long   cbMsg;
```

Where

pMsg

pointer to message text

cbMsg

length of message text

Returns

None, does not return.

Note

The code used to display the message is primitive. The message should contain ASCII characters in the range 0x20-0x7E, optionally with 0x0D and 0x0A to break the text into multiple lines.

The FSD must preface all such messages with the name of the file system.

Maximum message length is 128 characters. Messages longer than this are truncated.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_ISCURDIRPREFIX - Test for a prefix of a current directory

Since the kernel manages the text of each current directory for each process, the FSD must disallow any modification of any directory that is either a current directory of some process or the parent of any current directory of some process. FSDs call FSH_ISCURDIRPREFIX to achieve this.

FSH_ISCURDIRPREFIX will take the supplied path name, enumerate all current directories in use and test to see if the specified path name is a prefix or is equal to some current directory.

```
int pascal FSH_ISCURDIRPREFIX (pName)
char *pName;
```

Where

pName

pointer to path name. The name must be in canonical form: no '.' or '..' components, uppercase, no metacharacters, and full pathname specified.

Returns

Error code if pName is the prefix of or equal to the current directory of a process, 0 otherwise.

ERROR_CURRENT_DIRECTORY - the specified path is a prefix of or is equal to the current directory of some process.

Note

If the current directory is the root and the pathname is "d:\", ERROR_CURRENT_DIRECTORY will be returned.

FSH_ISCURDIRPREFIX may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_LOADCHAR - Load a character from a string

Provides mechanism for loading a character from a string taking into account DBCS considerations.

```
void pascal FSH_LOADCHAR (ppStr, pChar)
char * *      ppStr;
unsigned long * pChar;
```

Where

ppStr

pointer to character pointer to string. The character at this location will be retrieved and this pointer will be updated.

pChar

pointer to character returned. If character is non-DBCS, the first byte will be the character and the second byte will be zero.

Note

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_LOCKRANGE - Lock or unlock a range of memory

Provides mechanism for locking or unlocking a range of memory.

```
int pascal FSH_LOCKRANGE(pMem, cbMem, flag)
char *      pMem;
unsigned long cbMem;
unsigned long flag;
```

Where

pMem

pointer to memory to lock

cbMem

number of bytes to lock

flag

indicates to lock/unlock range

flag == 0 indicates lock range

flag == 1 indicates unlock range

Returns

Error code if error detected, 0 otherwise.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_PREVCHAR - Decrement character pointer

Provides mechanism for decrementing a character pointer taking into account DBCS considerations.

```
void pascal FSH_PREVCHAR (pBeg, ppStr)
char *      pBeg;
char * *    ppStr;
```

Where

pBeg

pointer to beginning of string

ppStr

pointer to character pointer. This value is decremented appropriately upon return. If it is at the beginning of the string, the pointer is not decremented. If it points to the second byte of a DBCS char, it will be decremented to point to the first byte of the char.

Note

The FSD is responsible for verifying the string pointer and checking for segment boundaries.

Reminder: OS/2 does not validate input parameters, so FSD should call

FSH_PROBEBUF - User address validity check

Since users may pass in arbitrary pointers to data, FSDs must perform validity checks on these pointers before using them. Because OS/2 is multithreaded, the addressability data returned by FSH_PROBEBUF is only valid until the FSD blocks. Blocking, either explicitly or implicitly allows other threads to run, and possibly invalidate a user segment. Therefore, FSH_PROBEBUF must be reapplied after every block.

FSH_PROBEBUF provides a convenient method for assuring that a user transfer address is valid and present in memory. Upon successful return, the user address may be treated as a pointer and accessed up to the specified length without either blocking or faulting. This state of affairs is guaranteed up until the FSD returns or until the next block.

Once FSH_PROBEBUF detects a protection violation, the process will be killed as soon as possible. The OS/2 kernel will kill the process once it has exited from the FSD.

FSH_PROBEBUF will insure that all touched pages are physically present in memory, so that the FSD will not suffer an implicit block due to a page fault. However, FSH_PROBEBUF does NOT guarantee that the pages will be physically contiguous in memory, since FSDs are not expected to do DMA.

```
int pascal FSH_PROBEBUF (operation, pData, cbData)
unsigned long operation;
char *      pData;
unsigned long cbData;
```

Where

operation

indicates whether read or write access is desired

operation == 0 indicates read access is to be checked.

operation == 1 indicates write access is to be checked.

All other values are reserved.

pData

starting address of user data

cbData

length of user data

Returns

Error code if either the access to the data is inappropriate or the user transfer region itself is partially or completely inaccessible. 0 otherwise.

ERROR_PROTECTION_VIOLATION - access to the indicated memory region is illegal.

Note

FSH_PROBEBUF may block.

To help avoid deadlocks, FSH_PROBEBUF should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_QSYSINFO - Query system information

FSH_QSYSINFO queries the system about dynamic system variables and static system variables not returned by DOSQSYSINFO.

```
int pascal FSH_QSYSINFO (index, pData, cbData)
unsigned long index;
char *      pData;
unsigned long cbData;
```

Where

index

variable to return

index == 1 indicates maximum sector size

pData

long address to data area

cbData

length of data area

Returns

Error code if error detected, 0 otherwise.

ERROR_INVALID_PARAMETER - invalid index.

ERROR_BUFFER_OVERFLOW - the specified buffer is too short for the returned data.

Note

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_RELEASEBUF - Release the owned buffer

FSH_RELEASEBUF releases a buffer owned by the calling thread if any is owned.

```
void pascal FSH_RELEASEBUF ()
```

Returns

Nothing

Note

See note under FSH_GETBUF for interactions with other buffer calls.

FSH_RELEASEBUF will not block.

FSH_REMOVESHARE - Remove a share entry

FSH_REMOVESHARE is used to remove a previously-entered name from the sharing set.

```
void pascal FSH_REMOVESHARE (hShare)
unsigned long hShare;
```

Where

hShare

share handle returned by a prior call to FSH_ADDSHARE.

Returns

None

Note

Once a call to FSH_REMOVESHARE has been issued, the share handle is no longer valid.

FSH_REMOVESHARE may block.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

To help avoid deadlocks, FSH_REMOVESHARE should not be called while owning an OS/2 buffer.

FSH_SEMCLR - Clear a semaphore

FSH_SEMCLR allows an FSD to release a semaphore that was previously obtained via FSH_SEMREQUEST.

```
int pascal FSH_SEMCLR (pSem)
char * pSem;
```

Where

pSem

handle to system semaphore or long address of ram semaphore

Returns

Error code if free failed, 0 otherwise.

ERROR_EXCL_SEM_ALREADY_OWNED - the exclusive semaphore is already owned.

ERROR_PROTECTION_VIOLATION - the semaphore is inaccessible.

Note

FSH_SEMCLEAR may block.

To help avoid deadlocks, FSH_SEMCLEAR should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_SEMREQUEST - Request a semaphore

FSH_SEMREQUEST allows an FSD to obtain exclusive access to a semaphore.

```
int pascal FSH_SEMREQUEST (pSem, cmsTimeout)
char *      pSem;
unsigned long cmsTimeout;
```

Where

pSem

handle to system semaphore or long address of ram semaphore

cmsTimeout

number of milliseconds to wait

Returns

Error code if failed, 0 otherwise.

ERROR_INTERRUPT - the current thread received a signal

ERROR_SEM_TIMEOUT - the timeout expired without gaining access to the semaphore.

ERROR_SEM_OWNER_DIED - the owner of the semaphore died.

ERROR_TOO_MANY_SEM_REQUESTS - there are too many semaphore requests in progress.

ERROR_PROTECTION_VIOLATION - the semaphore is inaccessible.

Note

The distinguished timeout value of 0xFFFFFFFF indicates an infinite timeout.

The caller may receive access to the semaphore after the timeout period has expired without receiving an ERROR_SEM_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

FSH_SEMREQUEST may block.

To help avoid deadlocks, FSH_SEMREQUEST should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_SEMSET - Set a semaphore

FSH_SEMSET allows an FSD to set a semaphore unconditionally.

```
int pascal FSH_SEMSET (pSem)
char * pSem;
```

Where

pSem

handle to system semaphore or long address of ram semaphore

Returns

Error code if invalid semaphore, 0 otherwise.

ERROR_EXCL_SEM_ALREADY_OWNED - the exclusive semaphore is already owned.

ERROR_TOO_MANY_SEM_REQUESTS - there are too many semaphore requests in progress.

ERROR_PROTECTION_VIOLATION - the semaphore is inaccessible.

Note

FSH_SEMSET may block.

To help avoid deadlocks, FSH_SEMSET should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_SEMSETWAIT - Set a semaphore and wait for clear

FSH_SEMSETWAIT allows an FSD to wait for an event. The event is signalled by a call to FSH_SEMCLEAR.

```
int pascal FSH_SEMSETWAIT (pSem, cmsTimeout)
char * pSem;
unsigned long cmsTimeout;
```

Where

pSem

handle to system semaphore or long address of ram semaphore

cmsTimeout

number of milliseconds to wait

Returns

Error code if timeout occurred or invalid semaphore, 0 otherwise.

ERROR_SEM_TIMEOUT - the timeout expired without gaining access to the semaphore.

ERROR_EXCL_SEM_ALREADY_OWNED - the exclusive semaphore is already owned.

ERROR_INTERRUPT - the current thread received a signal

ERROR_PROTECTION_VIOLATION - the semaphore is inaccessible.

Note

The caller may return after the timeout period has expired without receiving an ERROR_SEM_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

FSH_SEMSETWAIT may block.

To help avoid deadlocks, FSH_SEMSETWAIT should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_SEMWAIT - Wait for clear

FSH_SEMWAIT allows an FSD to wait for an event. The event is signalled by a call to FSH_SEMCLEAR.

```
int pascal FSH_SEMWAIT (pSem, cmsTimeout)
char *          pSem;
unsigned long cmsTimeout;
```

Where

pSem

handle to system semaphore or long address of ram semaphore

cmsTimeout

number of milliseconds to wait

Returns

Error code if free failed, 0 otherwise.

ERROR_SEM_TIMEOUT - the timeout expired without gaining access to the semaphore.

ERROR_INTERRUPT - the current thread received a signal

ERROR_PROTECTION_VIOLATION - the semaphore is inaccessible.

Note

The caller may return after the timeout period has expired without receiving an ERROR_SEM_TIMEOUT. Therefore, semaphore timeout values should not be used for exact timing and sequencing.

FSH_SEMWAIT may block.

To help avoid deadlocks, FSH_SEMWAIT should not be called while owning an OS/2 buffer.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_STORECHAR - Store a character in a string

Provides mechanism for storing a character into a string taking into account DBCS considerations.

```
void pascal FSH_STORECHAR (chDBCS, ppStr)
```

```
unsigned long chDBCS;
char * *      ppStr;
```

Where

chDBCS:

character to be stored. This may be either a single byte character or a double byte character with the first byte occupying the low-order position.

ppStr

pointer to character pointer where character will be stored. This pointer is updated upon return.

Note:

The FSD is responsible for verifying the string pointer and checking for segment boundaries.

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_WILDMATCH - Match using OS/2 wildcards

Provides mechanism for using OS/2 wildcards semantics to form a match between an input string and a pattern, taking into account DBCS considerations.

Wildcards provide a general mechanism for pattern matching file names. There are two distinguished characters that are relevant to this matching. The '?' character matches one character (not byte) except at a '.' or the end of a string, where it matches zero characters. The '*' matches zero or more characters (not bytes) with no implied boundaries except the end-of string. See discussion of meta characters.

For example, "a*b" matches "ab" and "aCCCCCCCCCb" while "a?b" matches "aCb" but does not match "aCCCCCCCCCb".

FSH_WILDMATCH is case-sensitive.

```
int pascal FSH_WILDMATCH (pPat, pStr)
char * pPat;
char * pStr;
```

Where

pPat

pointer to asciiz pattern string. Wildcards are present and are interpreted as follows: '?' will match exactly one character. '*' will match 0 or more characters. All other characters will be compared.

pStr

pointer to test string.

Returns

Error code if match failed, 0 otherwise.

ERROR_NO_META_MATCH - the wildcard match failed.

Note

Reminder: OS/2 does not validate input parameters, so FSD should call FSH_PROBEBUF where appropriate.

FSH_YIELD - Yield CPU to higher priority threads

Provides mechanism for relinquishing the processor to higher-priority threads. FSDs run under the 2ms dispatch latency imposed on the OS/2 kernel, meaning that no more than 2ms can be spent in an FSD without an explicit block or yield. FSH_YIELD will test to see if another thread is runnable at the current thread's priority or at a higher priority. If one exists, that thread will be given a chance to run.

```
void pascal FSH_YIELD ()
```

Returns

None.

System File Table

SCCSID=@(#)sft.doc 13.1 88/11/15

Changes needed:

All functions need to be modified to take an additional argument, a pointer to a TCB variable save area. The current implementation uses a static save area, eliminating any chance of re-entrancy. This additional parameter need to be passed on to the TCBSave and TCBrestore routines, as well.

All the calls to workers are now calls to DOS_Xxxx. With the advent of 1.2, these need to change to calls to FSD_Xxxx. The parameters are similar if not identical, so this should not be a problem. Note that if the Net installs early enough so that I can open the swap file at init time, paging over the net will fall out for free.

Currently there is no SFTresize function, so one needs to be added. This should not present a major problem since the code can be stolen straight out of w_newsize. It's sleazy and disgusting code, but is due to become very simple in 1.2.

I need to figure out if the flags passed to SFTopen are adequate, or if I need to play games to get the sharing modes to work.

Cdecl glue routines should be written, so that the functions can be called from kernel 32-bit C code. This should not be a problem, since the routines will be very simple.

The list of TCB variables modified by the file system may have changed since Greg did his work. Probably the safest thing to do is to save essentially everything initially, then pare it down as we merge with the 1.2 FS changes.

Exported functions:

SFTopen

```
takes(file name, sharing mode)
yields(SFN)
```

```
call TCBSave
Allocate an SFT entry from the system pool (via SFNFree)
call DOS_Open
call TCBrestore
```

SFTclose

```
takes(SFN)
yields(none)
```

```
call TCBSave
Call DOS_Close to close the SFT associated with SFN.
Should call SFLinkFree to return the now free SFT entry to the system
pool.
call TCBrestore
```

SFTread

```
takes(SFN, nbytes, transfer addr)
yields(nbytes)
```

```
call TCBSave
call SegLock to lock the segment
Calls DOS_Read
call SegUnlock
call TCBrestore
```

SFTwrite

```
takes(SFN, nbytes, transfer addr)
yields(nbytes)
```

```
call SFFFromSFN
call TCBSave
call SegLock to lock the segment
call DOS_Write
call SegUnlock
call TCBrestore
```

SFTseek

```
takes(SFN, dword offset)
yields(carry)
```

```
call SFFFromSFN
call TCBSave
call sft_lseek
call TCBrestore
```

SFTresize

```
takes(SFN, dword newsize)
yields(carry)
```

```
call SFFFromSFN
call TCBSave
sleaze wildy, or call FS_newsize in 1.2
call TCBrestore
```

Program Loader

This page is intentionally left blank.

NE - Microsoft Segmented Executable Format Description

Microsoft Segmented Executable Format

Version @ (#)1.8, 11/06/87

This is the "NE" format, as used by OS/2 1.x 16-bit executables. The OS/2 2.x executable format is the "LX" format, as described in OS220EXE.DOC.

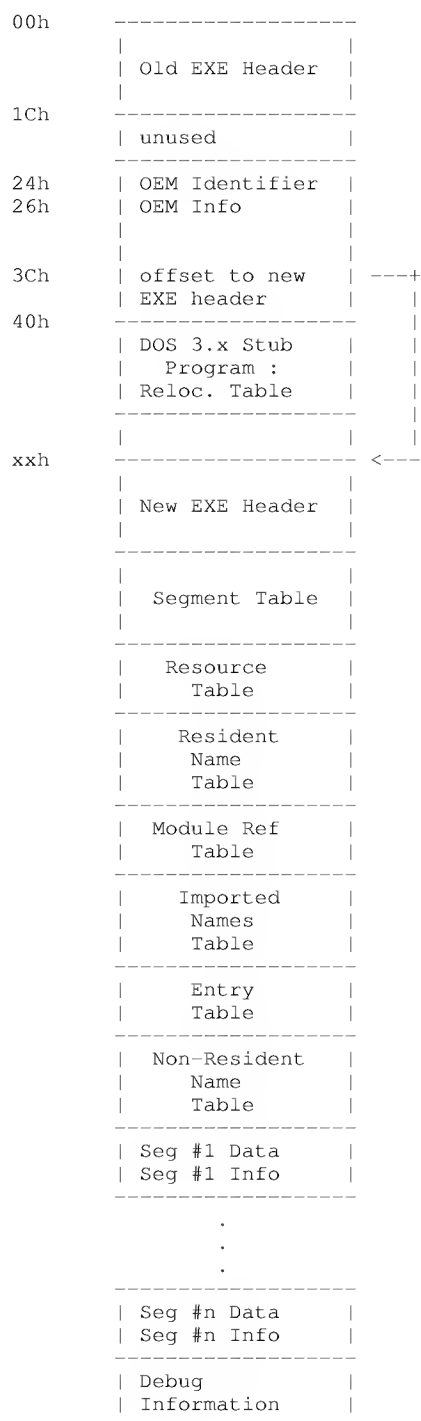
This document defines the Segmented Executable format, which is a superset of the MS-DOS 2.x and 3.x .EXE format (hereafter called DOS 3.x). The purpose of the segmented-executable format is to provide the information needed to support the dynamic linking and segmentation capabilities provided by OS/2, MS-DOS 4.0, and MS Windows. Also, information needed for protected mode is provided.

The DOS 3.x .EXE format has been extended as follows:

- The word at offset 18h in the existing .EXE file contains the relative byte offset to the relocation table. If this offset is 40h, then the double word at offset 3Ch is assumed to be the relative byte offset from the beginning of the file to the beginning of the new format executable header. A new format .EXE file is identified if the new executable header contains a valid signature. If the signature is not valid the file is assumed to be an old format .EXE file. The remainder of the old format header will describe a DOS 3.x program, the stub. The stub may be any valid program but will typically be a program which displays an error message, emulates the new-format program, or brings in a loader that can handle the job. See the picture below for the actual file layout.
- This format will only be used for .EXE files that use the new memory model supported by OS/2, MS-DOS 4.0 and MS Windows. Old .EXE files will continue with the DOS 3.x file format.

Throughout this specification, all unused fields and flag bits are reserved for future use and are expected to contain 0.

The new .EXE file has the following format:



New Fields in the Old Executable Header =====

```

24h  DW  OEM Identification
      A unique identifier assigned to an OEM to indicate
      ownership of the OEM Information definitions.
26h  DW  OEM Reserved Information
      Space reserved for OEM defined information.
28h  -
3Ah  DW  reserved for more behavior info
3Ch  DD  New EXE File Header Offset

```

New Executable Header =====

```

00h  DW  signature word
      "N" is low order byte
      "E" is high order byte
02h  DB  version#
03h  DB  revision#
04h  DW  Entry Table file offset relative to beginning of new EXE header
06h  DW  #bytes in Entry Table
08h  DD  32-bit checksum of entire contents of file
      (with these words taken as 00 during the calculation)
0Ch  DW  flag word
      0000h = NOAUTODATA
      0001h = Shared automatic data segment (SINGLEDATA)
      0002h = Instance automatic data segment (MULTIPLIEDATA)
      0004h = Per-process library initialization (INITINSTANCE)
      0008h = Runs in protected mode only (PROTMODE)
      0010h = 8086 instructions present
      0020h = 286 instructions present
      0040h = 386 instructions present
      0080h = Floating point instructions present
      0700h = Application type mask
            (P.M. = Presentation Manager)
      0100h = Not compatible with P.M. windowing
      0200h = Compatible with P.M. windowing
      0300h = Uses P.M. windowing API
      0800h = Bound Family/API program (set by BIND)
      2000h = Errors detected at link time
      8000h = Library module (SS:SP info is invalid, CS:IP points
            to initialization procedure that is called with AX
            = the module handle. The procedure must execute a
            far return to the caller, with AX != 0 to indicate
            success and AX = 0 to indicate failure to initialize.
            DS = the library's data segment if the SINGLEDATA
            flag is set and the caller's DS otherwise.)

```

A program can only contain dynamic links to executables that have this flag set.

```

0Eh  DW  segment# of automatic data segment (index into segment table)
      set to zero if SINGLEDATA and MULTIPLIEDATA flag bits are reset
10h  DW  initial size of dynamic heap added to data segment in bytes
      (0 if no local alloc)
12h  DW  initial size of stack added to data segment in bytes
      (0 if SS!=DS)
14h  DD  segment#&offset of CS:IP
18h  DD  segment#&offset of SS:SP
      Segment# is an index into the module's segment table.
      The first entry in the segment table is segment number 1.
      If SS = automatic data segment and SP = 0,
      the stack pointer is set to the top of the automatic
      data segment just below the additional heap area.

```

```

+-----+
| additional dynamic heap |
+-----+ <- SP
|   additional stack   |
+-----+
| loaded data segment  |
+-----+ <- DS, SS

```

```

1Ch  DW  #of entries in Segment Table
1Eh  DW  #of entries in Module Ref Table
20h  DW  #bytes in Non-Resident Name Table
22h  DW  Segment Table file offset relative to beginning of new EXE
      header
24h  DW  Resource Table file offset relative to beginning of new EXE
      header
26h  DW  Resident Name Table file offset relative to beginning of

```

```

new EXE header
28h  DW  Module Ref Table file offset relative to beginning of new
      EXE header
2Ah  DW  Imported Names Table file offset relative to beginning of
      new EXE header
2Ch  DD  Non-Resident Name Table offset relative to beginning of file
30h  DW  #of movable entries in Entry Table
32h  DW  logical sector alignment shift count, log(base 2) of segment
      sector size (default 9)
34h  DW  #of resource entries
36h  DB  Executable type
      00h = UNKNOWN
      01h = OS2
      02h = WINDOWS
      03h = DOS4
37h  -
3Eh  DW  reserved, currently 0's

```

Segment Table =====

"N" segment table entries:

```

The first entry in the segment table is segment number 1.
DW  n-byte logical sector offset to contents of the segment data
      relative to beginning of file (zero means no file data)
DW  length of segment in file, in bytes (zero means 64K)
DW  flag word
      0007h = TYPE_MASK      ; segment type field
      0000h = CODE          ; code segment type
      0001h = DATA         ; data segment type
      0008h = ITERATED      ; segment data is iterated
      0010h = MOVABLE       ; segment is movable (OS/2 ignores)
      0020h = SHARED        ; segment can be shared
      0040h = PRELOAD       ; segment is preloaded
      0080h = ERONLY        ; execute only if code segment
                          ; read only if data segment
      0100h = RELOCINFO     ; set if segment has reloc records
      0200h = CONFORM       ; segment is conforming
      0C00h = SEGDP1        ; I/O privilege level
      1000h = DISCARD       ; discardable segment (OS/2 ignores)
      4000h = HUGE          ; huge segment: length of segment and
                          ; minimum allocation sizes are in
                          ; units of segment sector size
DW  minimum allocation size in bytes
      Total size of the segment 0 means 64K

```

Resource Table =====

DW alignment shift count for resource data

"N" iterations of record:

```

| DW  type ID - integer type if high order bit is set (8000h)
|     otherwise offset to type string, relative to
|     beginning of the resource table
|     = 0 marks end of resource records
|
| DW  #resources for this type
| DD  Reserved
|
|     "#resources" copies of Resource Entry (8 bytes)
|
| DW  file offset to contents of the resource data relative
|     to beginning of file. Offset is in terms of alignment
|     units specified at beginning of resource table.
| DW  length of resource in file (bytes)
| DW  flag word
|     0010h = MOVEABLE      ; resource is not fixed
|     0020h = PURE         ; resource can be shared
|     0040h = PRELOAD       ; resource is not demand loaded
| DW  resource ID - integer type if high order bit is set (8000h)
|     otherwise offset to resource string, relative to beginning
| DD  Reserved
\   \   of the resource table

```

Resource type and name strings stored at end of resource table
Note that these strings are NOT null terminated

```

DB  length of type or name      ; = 0 if end of resource table
DB  ASCII text of type or name  ; Case sensitive

```

Resident or Non-resident Name Table Entry (3 + n bytes)

=====

The strings are CASE SENSITIVE and NOT NULL TERMINATED

DB Length of string ; =0 if no more strings in table
 DB ASCII text of string
 DW ordinal# (index into entry table)

First string in resident name table is the module name.

First string in non-resident name table is the module description.

Module Reference Table

=====

"N" entries of the form: (1-based)

DW offset within Imported Names Table to module name string

Imported Names Table (1 + n bytes)

=====

The strings are CASE SENSITIVE and NOT NULL TERMINATED

DB 0 ; to ensure non-zero first entry offset

"N" entries of the form:

DB Length of name
 DB ASCII text of name

Entry Table (1 based)

=====

"N" bundles of entry definitions. The ordinal value of an entry point is its ordinal within the entry table, counting the first entry as ordinal #1. The loader must scan over the bundles until it finds the bundle containing the entry point; the loader can then multiply by entry size to index the proper entry.

The linker forms bundles in the densest manner it can, given the restriction that it cannot reorder entry points to improve bundling because other EXE files may refer to entry points within this one by their ordinal in this table.

DB #entries in this bundle. All records in one bundle are either movable or refer to the same fixed segment. Equal to 0 if no more bundles in Entry Table.

DB segment indicator for this bundle
 | 00h - Unused, bundle not present, next bundle count follows
 | FFh - Movable segment, # is in entry
 | otherwise is segment # of fixed segment

If fixed segment, entries are 3 bytes:
 DB flags
 | 01h = set if entry is exported
 | 02h = set if entry uses global (shared) data segment
 | "mov ax,#ds-value" must be the 1st instruction in the prolog of this entry. This flag may only be set for SINGLEDATA library modules.
 | F8h = # of parameter words

Else movable segment, entries are 6 bytes:
 DB flags
 | 01h = set if entry is exported
 | 02h = set if entry uses global (shared) data segment
 | F8h = # of parameter words

int 3Fh
 DB segment#
 \ \ DW offset

Per segment data:

=====

```

If ITERATED
    DW #iterations
    DW #bytes of data
    DB data bytes
else
    DB data bytes

If RELOCINFO
    DW #relocation items
    |
    | Relocation Item: (8 bytes)
    |
    | DB source type
    |     0Fh = SOURCE_MASK
    |     00h = LOBYTE
    |     02h = SEGMENT
    |     03h = FAR_ADDR      (32-bit pointer)
    |     05h = OFFSET        (16-bit offset)
    |
    | DB flags
    |     03h = TARGET_MASK
    |     00h = INTERNALREF
    |     01h = IMPORTORDINAL
    |     02h = IMPORTNAME
    |     03h = OSFIXUP
    |     04h = ADDITIVE
    |
    | DW offset within this segment of source chain
    |     If ADDITIVE flag set, then add target value to source contents,
    |     instead of replacing source and following the chain.
    |     The source chain is a FFFFh terminated linked list within
    |     this segment of all references to the target.
    |
    | Target
    |     INTERNALREF
    |         DB segment# for fixed segment or FFh if movable
    |         DB 0
    |         DW offset into segment if fixed
    |             index into Entry Table iff movable
    |
    |     IMPORTNAME
    |         DW index into module ref table
    |         DW offset within Imported Names Table to proc. name string
    |
    |     IMPORTORDINAL
    |         DW index into module ref table
    |         DW procedure ordinal#
    |
    |     OSFIXUP
    |         DW Operating system fixup type
    |             Floating-point fixups
    |                 0001h = FIARQQ, FJARQQ
    |                 0002h = FISRQQ, FJSRQQ
    |                 0003h = FICRQQ, FJCRQQ
    |                 0004h = FIERQQ
    |                 0005h = FIDRQQ
    |                 0006h = FIWRQQ
    |
    | \ DW 0

Debug Information
=====
Any debug information will be at the end of the executable.
No information about it will appear in the header, so executable
files may be longer than the file length indicated in the
header.

```

LX - Linear eXecutable Module Format Description

This page is intentionally left blank.

LE Design History

SCCSID = @(#)ldrlefmt.doc 11.25 90/08/24

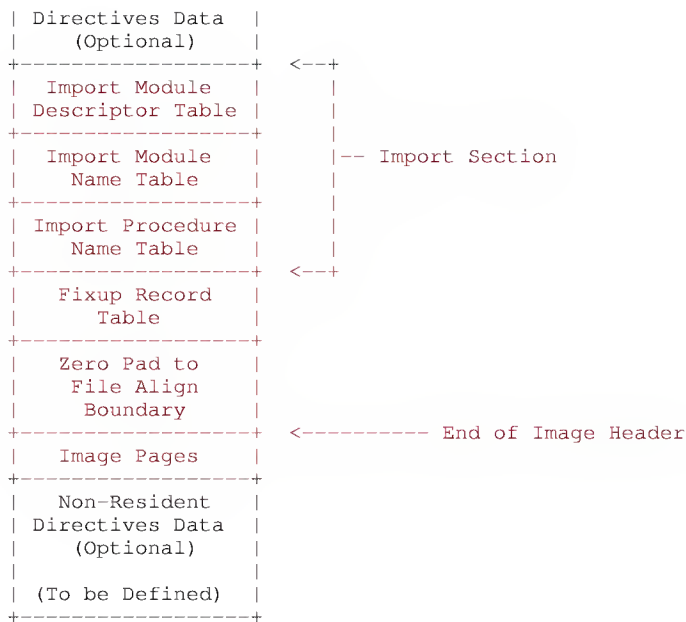
This is a redundant document. This is the 32-bit LE format that was not released. It was superseded by the LX format described in OS220EXE.DOC.

Revision codes:

revision 1 - Library termination (implemented).
revision 3 - Guard page stack support (planned).
revision 4 - NT enhancements (design stage).
revision 5 - NLS resources DCR (DCR in progress).

Introduction

00h	+-----+<--+<-----	Base of Image Header
	DOS 2 Compatible	
	EXE Header	
1ch	+-----+<--+<-----	
	unused	
	+-----+<--+<-----	
24h	OEM Identifier	
26h	OEM Info	
	+-----+<--+<-----	-- DOS 2.0 Section
3ch	Offset to	(for DOS compatibilty only)
	LE Module Header	
40h	+-----+<--+<-----	
	DOS 2.0 Stub	
	Program :	
	Reloc. Table	
	+-----+<--+<-----	
	unused	
	+-----+<-----	aligned on 4 byte boundary
	LE Module Header	
	+-----+<--+<-----	
	Object Table	
	+-----+<--+<-----	
	Language Table	
	+-----+<--+<-----	
	Resource Table	
	+-----+<--+<-----	
	Export Addr Table	
	+-----+<--+<-----	
	Export Auxiliary	
	Data Table	
	(To be Defined)	
	+-----+<--+<-----	
	Export Name	-- Export Section
	Table	
	+-----+<--+<-----	
	Export Name	
	Table Pointers	
	+-----+<--+<-----	
	Export Ordinal	
	Table	
	+-----+<--+<-----	
	Module Format	
	Directives Table	
	(Optional)	
	+-----+<--+<-----	
	Resident	



32-bit Linear EXE File Layout

32-bit Linear EXE Module Header

00h	"L"	"E"	BWORD	RESVD	MAJOR FMT	MINOR FMT
08h	CPU TYPE	OS TYPE	FILE CHECKSUM			
10h	MODULE FLAGS	RESERVED				
18h	MODULE MAJOR VERSION	MODULE MINOR VERSION				
20h	OBJECT TABLE OFFSET	LANGUAGE TABLE OFFSET				
28h	RESOURCE TABLE OFFSET	ORDINAL BASE				
30h	EXPORT ADDR TABLE OFF	EXPORT AUX DATA TBL OFF				
38h	EXPORT NAME TBL OFF	EXPORT NAME PTRS OFF				
40h	EXPORT ORD TBL OFFSET	MODULE DIRECTIVES OFF				
48h	# MODULE DIRECTIVES	IMPORT MOD DESC TBL OFF				
50h	IMPORT MOD NAME TBL OFF	IMPORT PROCNAME TBL OFF				
58h	IMPORT ADDR TABLE OBJ #	FIXUP REC TBL OFFSET				
60h	IMAGE PAGES OFFSET	IMAGE PAGES SIZE				
68h	PAGE SIZE	FILE ALIGN				
70h	VIRTUAL BASE	VIRTUAL SIZE				
78h	ENTRY VIRTUAL ADDR	ENTRY OBJECT #				
80h	STACK SIZE	STACK OBJECT #	*			
88h	HEAP SIZE	AUTO DATA OBJECT #	*	*		

* These fields are used only for 16-bit support

Module Header format

Notes:

1. Unless otherwise specified, table offsets in the Module Header are relative to the start of the file. If a table does not exist in the EXE file, its offset must be set to the offset of the following table, the sequence of tables is defined in the above figure, titled "32-bit Linear EXE File Layout". This allows a table's size to be computed by subtracting the table's offset from the offset of the following table, even when one or both tables is empty. Table offsets may not be set to zero to indicate that the table does not exist.
2. Unless otherwise specified, table offsets NOT in the Module Header are relative to the start of the appropriate table. Assuming a "mapped" EXE image, to compute the virtual address of an import name from an offset into the import name table, add all of the following together:

Virtual Address base for the image (from Module Header)
Import Procedure Name Table Offset (from Module Header)
Import Procedure Name Offset
3. Image pages are aligned and trailing zero truncated on File Align boundaries. The bases of all other tables and structures must be aligned on DWORD (4 byte) boundaries. All table and structure fields must be aligned on their "natural" boundaries, with the possible exception of the Debug Info.

Module Header Fields:

1. "L", "E" = DB * 2 Signature bytes. The signature bytes are used by the loader to identify the EXE file as a valid 32-bit Linear Executable Module Format. They are stored as a two byte character array to be independent of the byte and word order used in the image. "L" is the first byte, and "E" is the second byte.
2. BWORD = DB Byte and Word Ordering. This byte contains a bitfield specifying the byte and word ordering for the linear EXE format. The values are:

01h - Big Endian Byte Ordering (else Little Endian).
02h - Big Endian Word Ordering (else Little Endian).
fch = Reserved - must be zero.
3. RESVD = DB Reserved - must be zero.
4. MAJOR FMT = DW Linear EXE Major Format level.
5. MINOR FMT = DW Linear EXE Minor Format level. The Linear EXE Major Format level is set to 0 and the Minor Format level is set to 1 for this version of the 32-bit linear EXE format; Minor level 0 will be rejected by the loader. Prior to shipping OS/2 2.0, the Major Format level will be set to 1, and the Minor Format level will be set to 0. Each incompatible change to the linear EXE format must increment the Major Format level, as this allows the system to recognize older EXE file versions so that an appropriate error message may be displayed if an attempt is made to load them.
6. CPU TYPE = DW Module CPU Type. This field specifies the type of CPU compatibility required by this module to run. The values are:

0000h - unknown
0001h - 80286
0002h - 80386
0003h - 80486
0004h - 80586
0020h - i860 (N10)
0021h - i??? (N11)
0040h - MIPS Mark I (R2000, R3000)
0041h - MIPS Mark II (R6000)
0042h - MIPS Mark III (R4000)
7. OS TYPE = DW Module OS Type. This field specifies the type of Operating system or environment required to run this module:

0000h - Unknown
0001h - Microsoft/IBM OS/2 2.0 (default)
0002h - Microsoft Windows
0003h - Microsoft MS-DOS 4.x
0004h - Microsoft Windows 386
0020h - NT native (makes direct NT calls)
0021h - POSIX compliant binary
8. FILE CHECKSUM = DD Checksum for entire file. Set to 0 by the linker.
9. MODULE FLAGS = DD Flag bits for the module. The module flag bits have the following definitions:

00000003h = Reserved - must be zero.
 00000004h = Per-Process (Instance) Library Initialization. Only valid for libraries. When this bit is set, the Entry Address for the module must be valid, because it is called when each process first attaches to the library.
 00000008h = Reserved - must be zero.
 00000010h = Resolved fixups have been removed. When this bit is set, each object must be loaded at the addresses specified in the Object Table Virtual Address fields. This bit is set only for program modules and non-relocatable library modules.
 000000e0h = Reserved - must be zero.

00000000h = Illegal - reserved for future use.
 00000100h = Incompatible with PM windowing.
 00000200h = Compatible with PM windowing.
 00000300h = Uses PM windowing API.
 00000400h = Illegal - reserved for future use.
 00000500h = Illegal - reserved for future use.
 00000600h = Illegal - reserved for future use.
 00000700h = Illegal - reserved for future use.
 00000700h = Application Type mask.

00001800h = Reserved - must be zero.
 00002000h = Module is not loadable. When the 'Module is not loadable' flag is set, it indicates that either errors were detected at link time or that the module is being incrementally linked and therefore can't be loaded.
 00004000h = Reserved - must be zero.

00000000h = Program module.
 00008000h = Library module.
 00010000h = Illegal - reserved for future use.
 00018000h = Illegal - reserved for future use.
 00020000h = Physical Device Driver module.
 00028000h = Virtual Device Driver module.
 00030000h = Illegal - reserved for future use.
 00038000h = Illegal - reserved for future use.
 00038000h = Module type mask.

3ffc0000h = Reserved - must be zero.
 40000000h = Per-Process (Instance) Library Termination. Only valid for libraries. When this bit is set, the Entry Address for the module must be valid, because it is called when each process detaches from the library.
 80000000h = Reserved - must be zero.

10. RESERVED = DD Reserved - must be zero.
11. MODULE MAJOR VERSION = DD Major version of the linear EXE module.
12. MODULE MINOR VERSION = DD Minor version of the linear EXE module. This is useful for differentiating between revisions of dynamic linked modules. The values are specified at link time by the user.
13. OBJECT TABLE OFFSET = DD Object Table offset.
14. LANGUAGE TABLE OFFSET = DD Language Table offset.
15. RESOURCE TABLE OFFSET = DD Resource Table offset.
16. ORDINAL BASE = DD First valid exported ordinal. This field specifies the starting ordinal number for the export address table for this module. Normally set to 1.
17. EXPORT ADDR TABLE OFF = DD Export Address Table offset.
18. EXPORT AUX DATA TABLE OFF = DD Export Auxiliary Data Table offset.
19. EXPORT NAME TBL OFF = DD Export Name Table offset.
20. EXPORT NAME PTRS OFF = DD Export Name Table Pointers offset.
21. EXPORT ORD TBL OFFSET = DD Export Ordinals Table offset. The number of entries in the Export Name Table, the Export Name Table Pointers array, and the Export Ordinal Table can be derived by subtracting the offset of the Export Name Table Pointers array from the offset of the Export Ordinal Table, and dividing the result by 4.
22. MODULE DIRECTIVES OFF = DD Module Format Directives Table offset.
23. # MODULE DIRECTIVES = DD Number of Module Format Directives in the Table. This field specifies the number of entries in the Module Format Directives Table.

24. IMPORT MOD DESC TBL OFF = DD Import Module Descriptor Table offset.
25. IMPORT MOD NAME TBL OFF = DD Import Module Name Table offset.
26. IMPORT PROCNAME TBL OFF = DD Import Procedure Name Table offset.
27. IMPORT ADDR TABLE OBJ # = DD Import Address Table object number. This specifies the object containing the Import Address Table. It must be a non-zero value for any module that contains imports.
28. FIXUP REC TBL OFFSET = DD Fixup Record Table Offset
29. IMAGE PAGES OFFSET = DD Image Pages Offset. This offset is relative to beginning of the EXE file, is byte granular, and is aligned on a multiple of the File Align field in the Module Header.
30. IMAGE PAGES SIZE = DD Total size of image pages. The total size of the image pages for the module. It is byte granular, and must be a File Align multiple.
31. PAGE SIZE = DD The size of one page for this module. This field specifies the page size used by the linear EXE format. For the initial 80386 version of this linear EXE format the page size is 4Kbytes. (The 4K page size is specified by a value of 4096 in this field.)
32. FILE ALIGN = DD Alignment factor used to align/truncate image pages. The alignment factor (in bytes) used to align the base of the image pages and to determine the granularity of per-object trailing zero truncation. Should be a multiple of the sector size used in all file systems likely to be encountered. 1k is a reasonable default, due to the large number of hard disks using this sector size. Larger alignment factors will cost more file space; smaller alignment factors will impact demand load performance, perhaps significantly. Of the two, wasting file space is preferable.
33. VIRTUAL BASE = DD Virtual base address of module. The virtual base address of the module, indicating the relocation base address of the Image Header used by the linker. Must be set to 64k for program modules, which have their resolved fixups removed and are non-relocatable.
34. VIRTUAL SIZE = DD Virtual size of entire image. The virtual size (in bytes) of the image, including the Image Header and all other objects (debug information, too). The total image Virtual Size must be a 64k multiple.
35. ENTRY VIRTUAL ADDR = DD Entry Address of module. The entry virtual address is relative to the Virtual Base field stored in the Module Header. The Entry Address is the starting address for program modules and the library initialization and library termination address for library modules.
36. ENTRY OBJECT # = DD The Object number containing the Entry Address. This specifies the object to which the Entry Address is relative. This must be a non-zero value for a program module to be correctly loaded. A zero value for a library module indicates that no library entry routine exists. If this value is zero, then both the Per-process Library Initialization bit and the Per-process Library Termination bit must be clear in the module flags, or else the loader will fail to load the module. Further, if the Per-process Library Termination bit is set, then the object to which this field refers must be a 32-bit object (i.e., the Big/Default bit must be set in the object flags; see below).
37. STACK SIZE = DD Stack size for module. This field is ignored for a library module. For program modules, it contains the size of a 'guard page stack' object automatically allocated during program load. The guard page stack object is allocated as a completely uninitialized memory object with only one page committed and accessible at the high end of the object. The second highest page is created as a committed guard page. The system-provided default exception handlers will automatically cause the stack to be grown as needed, as long as sufficient memory resources are available.
38. STACK OBJECT # = DD The Object number containing the stack. This field is ignored for a library module. A non-zero value specifies the object to which the starting stack offset is relative, and indicates that the stack offset is stored in the stack size field. The stack offset defines the starting stack pointer address for program modules, and is relative to the start of the object containing the stack. A zero value in the stack size (stack offset) field indicates that the stack pointer is to be initialized to the highest address/offset in the object.

This field is supported for 16-bit compatibility only; it must be set to zero for 32-bit modules.
39. HEAP SIZE = DD Heap size added to the Auto DS Object. The heap size is the number of bytes added to the Auto Data Segment by the loader.

This field is supported for 16-bit compatibility only; it must be set to zero for 32-bit modules.
40. AUTO DATA OBJECT # = DD The Auto Data Segment Object number. This is the object number for the Auto Data Segment used by 16-bit modules.

This field is supported for 16-bit compatibility only; it must be set to zero for 32-bit modules.

Object Table

The number of entries in the Object Table is given by the # Objects in Module field in the Module Header. Entries in the Object Table are numbered starting from zero. The first Object Table entry (indexed by zero) maps the Image Header, up to the "Zero Pad to File Align Boundary". All code and data memory object entries follow the header entry, in the order chosen by the linker. If the linker resolved any fixups involving call gates (16-bit support only), then the next object table entry reserves virtual address space for the call gates needed. All resource memory objects follow. If debug info is present, it is mapped by the last Object Table entry, and is stored in the image pages, the same as for any other object. The image pages offsets and virtual addresses for objects must be assigned by the linker such that they are in ascending order and adjacent.

Each Object Table entry has the following format:

00h	VIRTUAL ADDR	VIRTUAL SIZE
08h	IMAGE PAGES OFFSET	IMAGE PAGES SIZE
10h	OBJECT FLAGS	RESERVED

Object Table

VIRTUAL ADDR = DD Relative Virtual Base Address. The virtual address the object is currently relocated to, relative to the Virtual Base field stored in the Module Header. The Relative Virtual Address of the first Object Table entry is zero, and maps the Image Header. Subsequent Object Table entries are based at the next available 64k boundary, so that each Object's virtual address space consumes a multiple of 64k, and immediately follows the previous Object in the virtual address space (the virtual address space for a module must be dense).

If the module is non-relocatable (as are all program modules), then resolved relocation fixups for the module have been removed, and this field added to the Virtual Base for the image is the address at which the object will be allocated by the loader.

VIRTUAL SIZE = DD Virtual memory size. The size of the object that will be allocated when the object is loaded. The object's Virtual Size is byte granular.

IMAGE PAGES SIZE = DD Physical file size of initialized data. The size of the initialized data in the file for the object. The physical size must be a multiple of the File Align field in the Module header, and must be less than or equal to the virtual size after rounding the virtual size up to a multiple of File Align.

If the physical size is not a page size multiple, the uninitialized page fragment is always zero filled. Any fully uninitialized pages at the end of an object are handled as zero fill or invalid pages by the loader, based on the "Trailing pages are invalid" bit in the object table flags field.

IMAGE PAGES OFFSET = DD Image Pages Offset for object's first page. This offset is relative to beginning of the EXE file, is byte granular, and is aligned on a multiple of the File Align field in the Module Header.

OBJECT FLAGS = DD Flag bits for the object. The object flag bits have the following definitions:

00000001h = Readable Object.
00000002h = Writable Object.
00000004h = Executable Object. The readable, writable and executable flags provide support for all possible protections. In systems where all of these protections are not supported, the loader will be responsible for making the appropriate protection match for the system.
00000008h = Resource Object.
00000010h = Discardable Object.
00000020h = Object is Shared.
00000040h = Reserved - must be zero.
00000080h = Trailing pages are invalid (else zero fill). Not allowed if the object is executable or shared.

00000000h = Object is NONPERMANENT/or normal EXE,DLL object.
00000100h = Object is PERMANENT (FSDs, VDDs, PDDs only).
00000200h = Object is RESIDENT (FSDs, VDDs, PDDs only).
00000300h = Object is CONTIGUOUS (FSDs, VDDs, PDDs only).
00000400h = Object is DYNAMIC (FSDs, VDDs, PDDs only).
00000500h = Illegal - reserved for future use.
00000600h = Object maps Image Header
00000700h = Object maps Debug Info
00000800h = Object reserves space for call gates.
00000900h - 00000f00h = Illegal - reserved for future use.
00000f00h = Object type mask

00001000h = 16:16 Alias Required (80x86 Specific).
00002000h = Big/Default Bit Setting (80x86 Specific). The 'big/default' bit, for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment

descriptor. (The Default bit, or D-bit, determines whether the default word size is 32-bits or 16-bits. It also affects the interpretation of the instruction stream.)
 00004000h = Object is conforming for code (80x86 Specific). Also used to indicate that data objects are expand down. Expand down data objects have the initialized data placed at the end of the object's virtual address space, and any implied zero fill or invalid pages precede the initialized data. Note that expand down objects are not supported by the program loader.
 00008000h = Object has I/O privilege level (80x86 Specific). Only used for 16:16 Alias Objects.
 00010000h = Object must not be cached. Only used on hardware that restricts certain operations on cached memory, such as locks on MIPS machines.
 ffe0000h = Reserved bits - must all be zero.

RESERVED = DD Reserved - must be zero.

Assuming the following:

- a program module
- a 1948 byte Image Header
- File Align set to 1024
- one code object (4097 bytes long)
- one data object (0 bytes initialized, 1 dword of zeros)
- one data object (1 byte initialized, 100k-1 bytes of zeros)
- two internal call gates assigned by the linker
- one resource object (1 byte)
- debug info (1 byte)

Then the Object Table would look like:

Obj#	Type	VirtAddr	VirtSize	ImageOffset	ImageSize	Flags
0	hdr	0	1948	0	2k	r+hdr
1	code	64k	4097	2k	5k	rx
2	data	2*64k	4	0	0	rw
3	data	3*64k	100k	7k	1k	rw
4	callgate	5*64k	128k	0	0	r+rsrv
5	resource	7*64k	1	8k	2k	r+rsrc
6	debug	8*64k	1	9k	2k	r+dbg

r - read permission
 w - read permission
 x - read permission
 hdr - mappable Image Header
 rsrv - call gate virtual address reservation
 rsrc - resources
 dbg - mappable HLL debug info

Object Table Example

Language Table

The language table is an array of language table entries. Every resource must be associated with a language table entry. If resources for a default language are supplied, the first language table entry must be the default language entry.

Each language entry has the following format:

00h	# RESOURCES	RESERVED	
04h	LANG ID	SUBLANG ID	
08h	RESOURCE TABLE OFFSET		

Language Table

RESOURCES = DW Resource Count for specified language.

RESERVED = DW Reserved - must be zero.

LANG ID = DW Language Identifier.

SUBLANG ID = DW Sublanguage Identifier.

RESOURCE TALE OFFSET = DD Resource Table Offset. This field contains an offset into the resource table for the resources for the specified language.

Resource Table

The resource table is an array of resource table entries. Each resource table entry contains a type ID and name ID which are used to locate resource objects contained in the Object table. **The number of entries in the resource table is computed by subtracting the Resource Table Offset from the Export Address Table Offset (both fields are located in the Module Header), and dividing by the size of a resource table entry.** A single object usually contains all resources for a given language. More than one object may be used if necessary for 16-bit compatibility, at the expense of internal page fragmentation for loaded resources. Resource table entries are in ascending sorted order by Resource Name ID within the Resource Type ID. This allows the DosGetResource API function to use a binary search when looking up a resource in a 32-bit module instead of the linear search being used in the current 16-bit module.

Each resource entry has the following format:

00h	TYPE ID	NAME ID	LENGTH
08h	CODEPAGE	OBJECT	VIRTUAL ADDR

Resource Table

RESOURCE TYPE ID = DW Resource type ID. A small subset of resource types:

- RT_POINTER = 1 Mouse pointer
- RT_BITMAP = 2 Bitmap
- RT_MENU = 3 Menu Template

RESOURCE NAME ID = DW Resource name ID.

CODEPAGE = DW Reserved - must be zero.

OBJECT = DW The number of the object which contains the resource.

VIRTUAL ADDR = DD Relative Virtual Address of the resource. The virtual address of the resource relative to the Virtual Base field stored in the Module Header.

Export Address Table

The Export Address Table contains address information that is used to resolve fixup references to the entry points within this module. Not all addresses in the table will be exported; private 16-bit call gate entries will only be used within the module. An ordinal number is used to index the Export Address Table. The address table entries are numbered starting from the value stored in the Ordinal Base field in the Module Header (usually one).

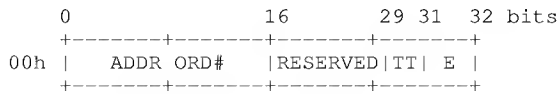
Export Address Table entry formats are described below:

0	31	32 bits
00h	EXPORT ADDRESS	E

Unescaped Export Address Table Format

EXPORT ADDRESS = 31-bits (mask = 7fffffffh) Export address. This field contains the 31 bit virtual address of the exported entry (relative to the Virtual Base stored in the Module Header).

E = 1-bit (mask = 80000000h) Escape bit. This bit is zero for unescaped Export Address Table entries.



Export Address Table Forwarder Format

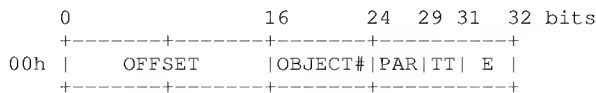
ADDR ORD# = 16-bits (mask = 0000ffffh) Import Address index. This field contains an index into the Import Address Table, which resides in the current module.

RESERVED = 13-bits (mask = 1fff0000h) Reserved - must be zero.

TT = 2-bits (mask = 60000000h) Entry Type.

00000000h = Forwarded Entry.
 20000000h = Exported Absolute/Call Gate Entry (see below).
 40000000h = Internal Call Gate Entry (see below).
 60000000h = Auxiliary Data Entry.

E = 1-bit (mask = 80000000h) Escape bit. This bit is set for all escaped Export Address Table entries.



Export Address Table Absolute/Call Gate Format

OFFSET = 16-bits (mask = 0000ffffh) Offset. If the OBJECT# field is non-zero, this field contains the offset of the entry point, relative to the start of the object containing the entry point for this Export Address Table entry's index (export ordinal number).

If the OBJECT# field is zero, this field contains the exported absolute value, and the entry type field must indicate Exported Absolute/Call Gate.

OBJECT# = 8-bits (mask = 00ff0000h) Object Table index. The object table index of the object containing the entry point for this Export Address Table entry's index.

PAR = 5-bits (mask = 1f000000h) Parameter word count. This field contains the parameter word count to be stored with the target's selector and offset in the call gate descriptor by the loader. When a ring 3 reference to a ring 2 entry point is made, the call gate selector with a zero offset is used to resolve relocation fixups.

TT = 2-bits (mask = 60000000h) Entry Type.

00000000h = Forwarded Entry (see above).
 20000000h = Exported Absolute/Call Gate Entry.
 40000000h = Internal Call Gate Entry.
 60000000h = Auxiliary Data Entry.

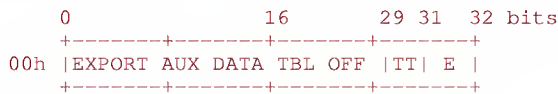
E = 1-bit (mask = 80000000h) Escape bit. This bit is set for all escaped Export Address Table entries.

A Forwarder entry is an entry point whose value is an imported reference. When a loadtime fixup occurs whose target is a forwarder, the loader obtains the address imported by the forwarder and uses that imported address to resolve the fixup.

A forwarder may refer to an entry point in another module which is itself a forwarder, so there can be a chain of forwarders. The loader will traverse the chain until it finds a non-forwarded entry point which terminates the chain, and use this to resolve the original fixup. Circular chains are detected by the loader and result in loadtime errors. A maximum of 1024 forwarders is allowed in a chain; more than this results in a loadtime error.

Forwarders are useful for merging and recombining API calls into different sets of DLLs, while maintaining compatibility with applications. For example, if one wanted to combine MONCALLS, MOUCALLS, and VIOCALLS into a single DLL, one could provide entry points for the three DLLs that are forwarders pointing to the common implementation.

Forwarders always refer to Import Address Table entries that contain the FLAT offset of the entry point. Fixup references to forwarders that involve non-FLAT pointers are supported by the loader computing the tiled address from the FLAT offset. The forwarder mechanism does not support forwarding to entry points requiring call gates. If forwarding to a call gate is required, the actual entry point should be a thunk in ring 3 (or ring 2 conforming code) that calls through an internal call gate to get to the ring 2 code.



Export Address Table Auxiliary Data Format

EXPORT AUX DATA TBL OFF = 29-bits (mask = 1fffff) Import Address index. This field contains an offset into the Export Auxiliary Data Table, which immediately follows the Export Address Table.

TT = 2-bits (mask = 60000000h) Entry Type.

- 00000000h = Forwarded Entry.
- 20000000h = Exported Absolute/Call Gate Entry.
- 40000000h = Internal Call Gate Entry.
- 60000000h = Auxiliary Data Entry (see below).

E = 1-bit (mask = 80000000h) Escape bit. This bit is set for all escaped Export Address Table entries.

Export Auxiliary Data Table.

The content of the Export Auxiliary Data Table is not yet defined. The initial implementation requires that the table be empty, and that no Auxiliary Data Entries be used in the Export Address Table.

Export Name Table Pointers

The export name table pointers array contains offsets into the Export Name Table. The pointers are 4 bytes each, and are relative to the start of the Export Name Table. The pointers are ordered lexically by byte code point to allow binary searches. Each byte in a name is treated as an unsigned numeric value for the purposes of sorting and comparing name strings. A zero pointer value is invalid, as it points to the module name (first name in the Export Name Table), and the module name cannot be exported. If an exported entry point name matches the module name, two identical strings will appear in the Export Name Table.

Export Ordinal Table

The Export Name Table Pointers and the Export Ordinal Table form two parallel arrays, separated to allow natural field alignment. The export ordinal table array contains the Export Address Table ordinal numbers associated with the named export referenced by corresponding Export Name Table Pointers.

The ordinals are 2 bytes each, and already include the Ordinal Base stored in the Module Header.

Export Name Table

The export name table contains optional ASCII names for exported entries in the module. In addition, the first entry in export name table contains the module name. These tables are used with the array of Export Name Table Pointers and the array of Export Ordinals to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the export address table.

The export name table is intended to contain the exported entry point names that are dynamic linked to by name. Exported entry point names that are never referenced via dynamic link by name should be eliminated. The trade off made for references by name is performance vs system memory usage and file size.

Import references by name require the Export Name Table Pointers table to be binary searched to find the matching name, then the corresponding Export Ordinal Table is known to contain the entry point ordinal number. Import references by ordinal number provide the fastest lookup since searching the name table is not required.

The strings are case sensitive and are terminated by a null byte.

Each name table entry has the following format:

```

+-----+-----+-----+-----+      +-----+
00h | ASCII STRING . . . | '\0' |
+-----+-----+-----+-----+      +-----+
```

Export Name Table Entry

ASCII STRING = DB ASCII String. The string is case sensitive and is terminated by a null byte. This is a variable length string with a maximum length of 127 bytes, including the trailing null byte.

Module Format Directives Table

The Module Format Directives Table is an optional table that allows additional options to be specified. It also allows for the extension of the linear EXE format by allowing additional tables of information to be added to the linear EXE module without affecting the format of the Module Header. Likewise, module format directives provide a place in the linear EXE module for 'temporary tables' of information, such as incremental linking information and statistic information gathered on the module. When there are no module format directives for a linear EXE module, the # Module Directives field in the Module Header is set to zero.

Each Module Format Directive Table entry has the following format:

```

+-----+-----+-----+-----+
00h | DIRECTIVE # | RESERVED |
+-----+-----+-----+-----+
04h |          DATA OFFSET          |
+-----+-----+-----+-----+
08h |          DATA LENGTH          |
+-----+-----+-----+-----+
```

Module Format Directive Table

DIRECTIVE # = DW Directive number. The directive number specifies the type of directive defined. This can be used to determine the format of the information in the directive data. The following directive numbers have been defined:

8000h = Resident Flag Mask. Directive numbers with this bit set indicate that the directive data is in the resident area and will be kept resident in memory when the module is loaded.

RESERVED = DW Reserved - must be zero.

DATA OFFSET = DD Directive data offset. This is the offset to the directive data for this directive number, and is relative to the beginning of the EXE file.

DATA LENGTH = DD Directive data length. This specifies the length in bytes of the directive data for this directive.

Fixup Record Table

The Fixup Record Table contains entries for all fixups in the linear EXE module.

Fixups that are resolved by the linker do not need to be processed by the loader, unless the load image is being relocated. Resolved fixups are ignored when a module is loaded at the Virtual Base address stored in the Module Header. All program module internal fixups are resolved by the linker. All library module internal fixups except internal call gate fixups are resolved by the linker. Program modules and non-relocatable library modules have all resolved fixups stripped by the linker. Through the use of thunks, many import fixups can be resolved by the linker, requiring only that the Import Address Table be filled in at load time.

The fixup records for each page are grouped together. If no fixups exist for a page, then no space is required for that page in the fixup record table.

All fixups are additive, meaning that any fixup (resolved or unresolved) may contain an additive value. The loader must always add in a address delta value for internal fixups, and add in the target value for import fixups.

Internal FLAT and non-FLAT 16:16 and 16:32 pointer fixups are split into two fixups, one for the selector, and one for the offset. Internal non-FLAT offset fixups are always completely resolved by the linker. Internal non-FLAT selector fixups and FLAT offset fixups are tentatively resolved by the linker (and eliminated if the image is non-relocatable). Internal FLAT selector fixups are tentatively resolved by the linker by storing 0 in the image page, and are always propagated to the load image.

Import non-FLAT 16:16 and 16:32 pointer fixups are maintained as a single fixup that points to a Import Address Table entry (for the non-FLAT 16:16 address). Import non-FLAT 16:32 pointer fixups are resolved to a 16:16 address by the loader, and the 16 bit offset is then zero extended to 32 bits before writing the result into the image page.

Note that import FLAT selector and FLAT pointer fixups are not supported.

Relocation records have one of the following formats:

```

+-----+-----+-----+-----+
00h | REL VA |E|      CNT      |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+
```

* These fields are variable size.

@ These fields are optional.

Unescaped Fixup Record Format

Unescaped fixups are used to represent the most common type of fixup in relocatable modules, internal FLAT offset fixups. If any fixups exist for a page, then the first fixup must be an unescaped internal FLAT offset fixup, which encodes the relative virtual address for the page. This allows the virtual address for each page to be specified prior to processing fixups for each page. If no internal FLAT offset fixups are required for a page that has other fixups, then the count of source offsets must be set to zero.

An unescaped fixup with a zero Relative Virtual Address and a zero source offset count is used to terminate the fixup record table.

Escaped fixups must be used for fixups that cross page boundaries, because the value used by the linker to resolve the fixup (as well as any additive value stored in the image pages by the linker) may not be available to the loader, and must be stored in the fixup record. Escaped fixups must be used for all fixups other than internal FLAT offset fixups.

E = 1-bit (mask = 8000h) Escape bit. This bit is clear for all unescaped fixup records.

REL VA = 15-bits (mask = 7fffh) Relative Virtual Address. Shifting the contents of this field left by 16 bits results in the virtual address of the page relative to the Virtual Base field stored in the Module Header.

CNT = DW Source offset list count. A list of source offsets follows the end of fixup record. This field contains the number of source offsets in the list, and may be zero to indicate no source offsets exist.

SRCOFF1 - SRCOFFn = DW[] Source offset list. The number of entries in the source offset list is defined in the CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.

```

+-----+-----+-----+-----+
00h |  FLAGS  |E|SRCOFF/CNT@|
+-----+-----+-----+-----+
04h |          TARGET INFO * @          |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+
```

* These fields are variable size.
 @ These fields are optional.

Escaped Fixup Record Format

E = 1-bit (mask = 8000h) Escape bit. This bit is set for all escaped fixup records.

FLAGS = 15-bits (mask = 7fffh) Fixup Flags. Source flags specify the size and type of the fixup to be performed on the fixup source. The source flags are defined as follows:

0000h = Byte fixup (8-bits).
 0001h = Align fixup - nop used to skip 2 bytes.
 0002h = 16-bit Selector fixup (16-bits).
 0003h = 16:16 Pointer fixup (32-bits).
 0004h = Illegal - reserved for future use.
 0005h = 16-bit Offset fixup (16-bits).
 0006h = 16:32 Pointer fixup (48-bits).
 0007h = 32-bit Offset fixup (32-bits).
 0008h = 32-bit Self-relative offset fixup (32-bits).
 0009h - 000fh = Illegal values - reserved for future use.
 000fh = Source mask.

0010h = Fixup to Alias Flag. When the 'Fixup to Alias' Flag is set, the source fixup refers to the 16:16 alias for the object. This is only valid for source types of 2, 3, and 6. For fixups such as this, the linker and loader will be required to perform additional checks such as ensuring that the target offset for this fixup is less than 64K.

0020h = Source List Flag. When the 'Source List' Flag is set, the SRCOFF field contains the number of source offsets, and a list of source offsets follows the end of fixup record.

0040h = Fixup Data Flag. When the 'Fixup Data' Flag is set, the FIXUP DATA field contains the 32-bit value written to the image page by the linker. This flag is only set when the fixup crosses a page boundary.

0080h = Source has IOPL and is not CONFORMING. Target flags specify how the target information is interpreted.

The target flags are defined as follows:

0000h = Internal reference.
 0100h = Imported reference by ordinal or name.
 0200h = Illegal - reserved for future use.
 0300h = Internal reference via export address table.
 0300h = Fixup target type mask.
 0400h = Target has IOPL and is not CONFORMING. Valid only for non-aliased (FLAT) selector fixups.
 0800h = Target is CODE (else DATA). Valid only for non-aliased (FLAT) selector fixups.
 f000h = Reserved - must be zero.

SRCOFF/CNT = DW Source offset or source offset list count. This field contains either an offset or a count depending on the Source List Flag. If the Source List Flag is set, a list of source offsets follows the fixup record and this field contains the count of the entries in the source offset list. Otherwise, this is the single source offset for the fixup. Source offsets are relative to the beginning of the page where the fixup is to be made.

Note that for fixups that cross page boundaries, a separate fixup record is specified for each page. An offset is still used for the 2nd page but it now becomes a negative offset since the fixup originated on the preceeding page. (For example, if only the last byte of a 32-bit address is on the page to be fixed up, then the source offset would have a value of -3.)

TARGET DATA = Target data for fixup. The format of the TARGET DATA is dependent upon target flags.

SRCOFF1 - SRCOFFn = DW[] Source offset list. This list is present if the Source List Flag is set in the Target Flags field. The number of entries in the source offset list is defined in the SRCOFF/CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.

```
+-----+-----+
00h |  FLAGS  |E|
+-----+-----+
```

Escaped Align Fixup Record

Align fixups should only be generated when the next fixup record would otherwise not be dword aligned (50% chance), and the next fixup contains the FIXUP DATA field (a rare event).

```
+-----+-----+-----+-----+
00h |  FLAGS  |E|SRCOFF/CNT |
+-----+-----+-----+-----+
04h |          FIXUP DATA @          |
+-----+-----+-----+-----+
04/08h | SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+
```

* These fields are variable size.
 @ These fields are optional.

Escaped Internal Fixup Record

FIXUP DATA = DD Fixup data as resolved by the linker. The FIXUP DATA is used only when a fixup crosses a page boundary, and the current value stored in the image page is needed to compute the new value to be written to the image page.

00h		FLAGS	E	SRCOFF/CNT	
04h		FIXUP DATA @			
04/08h		ADDR	ORD#		
06/0ah		SRCOFF1 @		. . .	SRCOFFn @

* These fields are variable size.
 @ These fields are optional.

Escaped Import by Name or Ordinal Fixup Record

FIXUP DATA = DD Fixup data as resolved by the linker. The FIXUP DATA is used only when a fixup crosses a page boundary, and the current value stored in the image page is needed to compute the new value to be written to the image page.

ADDR ORD # = DW Import Address Table index. This field contains an index into the Import Address Table, which resides in the current module.

00h		FLAGS	E	SRCOFF/CNT	
04h		FIXUP DATA @			
04/08h		ORD #			
06/0ah		SRCOFF1 @		. . .	SRCOFFn @

* These fields are variable size.
 @ These fields are optional.

Escaped Internal Export Address Table Fixup Record

FIXUP DATA = DD Fixup data as resolved by the linker. The FIXUP DATA is used only when a fixup crosses a page boundary, and the current value stored in the image page is needed to compute the new value to be written to the image page.

ORD # = DW Ordinal index into the Export Address Table. This field is an index into the current module's Export Address Table to specify the target address. The Export Address Table entry must involve a call gate.

Import Module Descriptor Table

The import module descriptor table contains information about each imported module. Imported modules are referenced via relocation fixups.

Each module descriptor table entry has the following format:

00h		MODULE NAME OFFSET			
04h		IMPORT TABLE VIRT ADDR			

Import Module Name Table

An empty Import Module Descriptor is appended to the array, which contains a valid import address table offset that points to the end of import address table, a zero modname name offset and a zero flags entry,

MODULE NAME OFFSET = DD Imported module name table offset. Offset into the Import Module Name Table for the name of the module being imported.

IMPORT TABLE VIRT ADDR = DD Import Address Table Virtual Address. This field contains the virtual address of the start of the import addresses for this imported module, relative to the Virtual Base field stored in the Module Header. The next Import Module Descriptor's import address table virtual address can be used to compute the size of the import address table for this imported module.

Import Module Name Table

The import module name table defines the module name strings imported by this module through dynamic link references. These strings are referenced via the Import Module Descriptor Table.

To determine the length of the import module name table, subtract the import module name table offset from the import procedure name table offset. These values are located in the Module Header.

The strings are case sensitive and terminated by a null byte. A zero offset is a valid name table offset, which references the first entry in the table.

Each module name table entry has the following format:

	+-----+-----+-----+-----+	+-----+
00h	ASCII STRING . . .	'\0'
	+-----+-----+-----+-----+	+-----+

Import Module Name Table

ASCII STRING = DB ASCII String. The string is case sensitive and is terminated by a null byte. This is a variable length string with a maximum length of 127 bytes, including the trailing null byte.

Import Procedure Name Table

The import procedure name table defines the procedure name strings imported by this module through dynamic link references. These strings are referenced via the Import Address Table.

To determine the length of the import name table, add the fixup section size to the fixup page table offset. This computes the offset to the end of the fixup section, then subtract the import procedure name table offset. These values are located in the Module Header.

The strings are case sensitive and terminated by a null byte. A zero offset is a valid name table offset, which references the first entry in the table.

Each name table entry has the following format:

	+-----+-----+	
00h	HINT	
	+-----+-----+-----+-----+	+-----+
02h	ASCII STRING . . .	'\0'
	+-----+-----+-----+-----+	+-----+
	PAD@	
	+-----+	

@ These fields are optional.

Import Procedure Name Table

HINT = DW ASCII String. The 16-bit hint value is used to index the import module's Export Name Table Pointers array, allowing faster by-name imports. In the case where the import module's exports change, the hint is used as a starting point for a binary search of the import module's Export Name Table. A hint value of 0 will cause a binary search of the entire import module's Export Name Table.

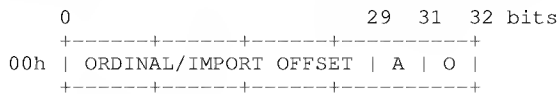
ASCII STRING = DB ASCII String. The string is case sensitive and is terminated by a null byte. This is a variable length string with a maximum length of 127 bytes, including the trailing null byte.

PAD = DB Zero pad byte. A trailing zero pad byte appears after the trailing null byte if necessary to align the next entry on an even boundary.

Import Address Table.

The linker appends the import address table to any handy use32 code object. If no use32 code object exists, a use16 code object is selected. If no code object exists, the linker creates a use32 code object.

Import Module Descriptors point to the import addresses for each import module. Relocation fixups index the Import Module Descriptor array and the Import Address Table for the appropriate import addresses.



Import Address Table Format

The loader initializes the import address table when loading the page(s) that contain it. During import address table initialization, the entire 32 bit field is overwritten with the 32-bit address of the import. The type of address needed is encoded in the address type field.

ORDINAL/IMPORT OFFSET = 29-bits (mask = 1fffffth) Ordinal number or import data offset. If the import is by ordinal, this field contains a 16 bit ordinal number. If the import is by name, this field contains a 29 bit offset into the Import Procedure Name Table.

A = 2-bits (mask = 60000000h) Address type flag.

00000000h = 0:32 Offset - Flat offset.
20000000h = 16:16 non-FLAT, non-gate pointer.
40000000h = 16:16 gate pointer - Callgate needed if used.
60000000h = Illegal - reserved for future use.

O = 1-bit (mask = 80000000h) Import by ordinal flag.

00000000h = Import by name.
80000000h = Import by ordinal.

Zero Pad to File Align Boundary

Since the image pages section is aligned on a boundary that is a multiple of the File Align field in the Module Header, padded space may exist between the end of the fixup section and the first page in the image pages section. If this padded space exists, it will be zero filled.

Image Pages

The Image Pages section contains all initialized data for all objects. The seek offset for the first page in each object is specified in the object table and is aligned on a File Align boundary. The pages are ordered by physical page number within this section. Every page begins on a page size offset from the image pages offset specified in the object table. All but the last page for each object are represented as full pages in the EXE file. The last page for each object may have full File Align multiples of trailing zeros removed.

Debug Information

The debug information is defined by the debugger and is not controlled by the linear EXE format or linker. The only data defined by the linear EXE format relative to the debug information is its offset in the EXE file and length in bytes as defined in the last object table entry.

To support multiple debuggers the first word of the debug information is a type field which determines the format of the debug information.

```
+-----+-----+-----+-----+
00h |   TYPE   |  DEBUGGER DATA  . . .
+-----+-----+-----+-----+
```

Debug Information

TYPE = DW Format type. This defines the type of debugger data that exists in the remainder of the debug information.

The following format types are defined.

0000h = CodeView debugger format.
0001h = AIX debugger format.

DEBUGGER DATA = Debugger specific data. The format of the debugger data is defined by the debugger that is being used.

The values defined for the type field are not enforced by the system. It is the responsibility of the linker and debugging tools to follow the convention for the type field that is defined here.

Program module (EXE) startup registers and Library (DLL) entry reg

OS/2 386 program startup registers are defined as follows:

EIP = Starting program entry address.
ESP = Top of stack address.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Pointer to the Thread Information Block (TIBSEL).
GS = 0.
EAX = EBX = 0.
ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.

[ESP+0] = Return address to routine which calls DosExit(1,EAX).
[ESP+4] = Module handle for program module.
[ESP+8] = Reserved.
[ESP+12] = Environment data object address.
[ESP+16] = Command line linear address in environment data object.

OS/2 386 library initialization registers are defined as follows:

EIP = Library entry address.
ESP = User program stack.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Pointer to the Thread Information Block (TIBSEL).
GS = 0.
EAX = EBX = 0.
ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.

[ESP+0] = Return address to system, (EAX) = return code.
[ESP+4] = 0 (Initialization)
[ESP+8] = Module handle for library module.

Note that a 32-bit library module may specify that its entry address is in a 16-bit code object. In this case, the entry registers are the same as for entry to a library module using the Segmented EXE format. These are documented elsewhere. This means that a 16-bit library module may be relinked to take advantage of the benefits of the Linear EXE format (notably, efficient paging).

OS/2 386 library module termination registers are defined as follows:

EIP = Library entry address.
ESP = User program stack.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Pointer to the Thread Information Block (TIBSEL).
GS = 0.
EAX = EBX = 0.
ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.

[ESP+0] = Return address to system.
[ESP+4] = 1 (Termination)
[ESP+8] = Module handle for library module.

Note that library termination is not allowed for library modules with 16-bit entries.

LX Design History

This page is intentionally left blank.

Revision codes

*** This is a redundant document. It has been superceeded by OS220EXE.DOC.**

revision 1 - Library termination (4/1/91).
revision 2 - Sector Align and Exepack support (GM-B).
revision 3 - Address Based linking (GM-C).
revision 4 - OS/2 2.0 PM Debugger (IBM) support (GM-B).

00h +-----+ <--+
| DOS 2 Compatible | |

	EXE Header		
1Ch	unused		
24h	OEM Identifier		
26h	OEM Info		
3Ch	Offset to Linear EXE Header		-- DOS 2.0 Section (Discarded)
40h	DOS 2.0 Stub Program : Reloc. Table		
		<--	
xxh	Executable Info	<--	
	Module Info		
	Loader Section Info		-- Linear Executable Module Header (Resident)
	Table Offset Info		
		<--	
	Object Table		
	Object Page Table		
	Resource Table		
	Resident Name Table		
	Entry Table		-- Loader Section (Resident)
	Module Format Directives Table (Optional)		
	Resident Directives Data (Optional)		
	(Verify Record)		
	Per-Page Checksum		
		<--	
	Fixup Page Table		
	Fixup Record Table		
	Import Module Name Table		-- Fixup Section (Optionally Resident)
	Import Procedure Name Table		
		<--	
	Preload Pages		
	Demand Load Pages		
	Iterated Pages		
	Non-Resident Name Table		-- (Non-Resident)
	Non-Resident Directives Data (Optional)		
	(To be Defined)		
		<--	
	Debug Info		-- (Not used by Loader)
		<--	

32-bit Linear EXE Header

00h	"L" "X"	B-ORD W-ORD	FORMAT LEVEL	
08h	CPU TYPE	OS TYPE	MODULE VERSION	
10h	MODULE FLAGS		MODULE # OF PAGES	
18h	EIP OBJECT #		EIP	
20h	ESP OBJECT #		ESP	
28h	PAGE SIZE		PAGE OFFSET SHIFT	
30h	FIXUP SECTION SIZE		FIXUP SECTION CHECKSUM	
38h	LOADER SECTION SIZE		LOADER SECTION CHECKSUM	
40h	OBJECT TABLE OFF		# OBJECTS IN MODULE	
48h	OBJECT PAGE TABLE OFF		OBJECT ITER PAGES OFF	
50h	RESOURCE TABLE OFFSET		#RESOURCE TABLE ENTRIES	
58h	RESIDENT NAME TBL OFF		ENTRY TABLE OFFSET	
60h	MODULE DIRECTIVES OFF		# MODULE DIRECTIVES	
68h	FIXUP PAGE TABLE OFF		FIXUP RECORD TABLE OFF	
70h	IMPORT MODULE TBL OFF		# IMPORT MOD ENTRIES	
78h	IMPORT PROC TBL OFF		PER-PAGE CHECKSUM OFF	
80h	DATA PAGES OFFSET		#PRELOAD PAGES	
88h	NON-RES NAME TBL OFF		NON-RES NAME TBL LEN	
90h	NON-RES NAME TBL CKSM		AUTO DS OBJECT #	
98h	DEBUG INFO OFF		DEBUG INFO LEN	
A0h	#INSTANCE PRELOAD		#INSTANCE DEMAND	
A8h	HEAPSIZE			

Note that the OBJECT ITER PAGES OFF must either be 0 or set to the same value as DATA PAGES OFFSET in OS/2 2.0. Ie., iterated pages are required to be in the same section of the file as regular pages.

32-bit Linear EXE Header

Note: Table offsets in the Linear EXE Header may be set to zero to indicate that the table does not exist in the EXE file and it's size is zero.

"L" "X" = DW Signature word. The signature word is used by the loader to identify the EXE file as a valid 32-bit Linear Executable Module Format. "L" is low order byte. "X" is high order byte.

B-ORD = DB Byte Ordering. This byte specifies the byte ordering for the linear EXE format. The values are:

00H - Little Endian Byte Ordering.
01H - Big Endian Byte Ordering.

W-ORD = DB Word Ordering. This byte specifies the Word ordering for the linear EXE format. The values are:

00H - Little Endian Word Ordering.
01H - Big Endian Word Ordering.

Format Level = DD Linear EXE Format Level. The Linear EXE Format Level is set to 0 for the initial version of the 32-bit linear EXE format. Each incompatible change to the linear EXE format must increment this value. This allows the system to recognize future EXE file versions so that an appropriate error message may be displayed if an attempt is made to load them.

CPU Type = DW Module CPU Type. This field specifies the type of CPU required by this module to run. The values are:

01H - 80286 or upwardly compatible CPU is required to execute this module.
02H - 80386 or upwardly compatible CPU is required to execute this module.
03H - 80486 or upwardly compatible CPU is required to execute this module.

OS Type = DW Module OS Type. This field specifies the type of Operating system required to run this module. The currently defined values are:

00H - Unknown (any "new-format" OS)
01H - OS/2 (default)
02H - Windows
03H - DOS 4.x
04H - Windows 386

MODULE VERSION = DD Version of the linear EXE module. This is useful for differentiating between revisions of dynamic linked modules. This value is specified at link time by the user.

MODULE FLAGS = DD Flag bits for the module. The module flag bits have the following definitions.

00000001h = Reserved for system use.
00000002h = Reserved for system use.
00000004h = Per-Process Library Initialization. The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Initialization is assumed. Setting this bit for an EXE file is invalid.
00000008h = Reserved for system use.
00000010h = Internal fixups for the module have been applied. The setting of this bit in a Linear Executable Module indicates that each object of the module has a preferred load address specified in the Object Table Reloc Base Addr. If the module's objects can not be loaded at these preferred addresses, then the relocation records that have been retained in the file data will be applied.
00000020h = External fixups for the module have been applied.
00000040h = Reserved for system use.
00000080h = Reserved for system use.
00000100h = Incompatible with PM windowing.
00000200h = Compatible with PM windowing.
00000300h = Uses PM windowing API.
00000400h = Reserved for system use.
00000800h = Reserved for system use.
00001000h = Reserved for system use.
00002000h = Module is not loadable. When the 'Module is not loadable' flag is set, it indicates that either errors were detected at link time or that the module is being incrementally linked and therefore can't be loaded.
00004000h = Reserved for system use.
00038000h = Module type mask.
00000000h = Program module. A module can not contain dynamic links to other modules that have the 'program module' type.
00008000h = Library module.
00018000h = Protected Memory Library module.
00020000h = Physical Device Driver module.
00028000h = Virtual Device Driver module.
40000000h = Per-process Library Termination. The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Termination is assumed. Setting this bit for an EXE file is invalid.

MODULE # PAGES = DD Number of pages in module. This field specifies the number of pages physically contained in this module. In other words, pages containing either enumerated or iterated data, or zero-fill pages that have relocations, not invalid or zero-fill pages implied by the Virtual Size in the Object Table being larger than the number of pages actually in the linear EXE file. These pages are contained in the 'preload pages', 'demand load pages' and 'iterated data pages' sections of the linear EXE module. This is used to determine the size of the page information tables in the linear EXE module.

EIP OBJECT # = DD The Object number to which the Entry Address is relative. This specifies the object to which the Entry Address is relative. This must be a nonzero value for a program module to be correctly loaded. A zero value for a library module indicates that no library entry routine exists. If this value is zero, then both the Per-process Library Initialization bit and the Per-process Library Termination bit must be clear in the module flags, or else the loader will fail to load the module. Further, if the Per-process Library Termination bit is set, then the object to which this field refers must be a 32-bit object (i.e., the Big/Default bit must be set in the object flags; see below).

EIP = DD Entry Address of module. The Entry Address is the starting address for program modules and the library initialization and Library termination address for library modules.

ESP OBJECT # = DD The Object number to which the ESP is relative. This specifies the object to which the starting ESP is relative. This must be a nonzero value for a program module to be correctly loaded. This field is ignored for a library module.

ESP = DD Starting stack address of module. The ESP defines the starting stack pointer address for program modules. A zero value in this field indicates that the stack pointer is to be initialized to the highest address/offset in the object. This field is ignored for a library module.

PAGE SIZE = DD The size of one page for this system. This field specifies the page size used by the linear EXE format and the system. For the initial version of this linear EXE format the page size is 4Kbytes. (The 4K page size is specified by a value of 4096 in this field.)

PAGE OFFSET SHIFT = DD The shift left bits for page offsets. This field gives the number of bit positions to shift left when interpreting the Object Page Table entries' page offset field. This determines the alignment of the page information in the file. For example, a value of 4 in this field would align all pages in the Data Pages and Iterated Pages sections on 16 byte (paragraph) boundaries. A Page Offset Shift of 9 would align all pages on a 512 byte (disk sector) basis. The default value for this field is 12 (decimal), which give a 4096 byte alignment. All other offsets are byte aligned.

FIXUP SECTION SIZE = DD Total size of the fixup information in bytes. This includes the following 4 tables:

- Fixup Page Table
- Fixup Record Table
- Import Module name Table
- Import Procedure Name Table

FIXUP SECTION CHECKSUM = DD Checksum for fixup information. This is a cryptographic checksum covering all of the fixup information. The checksum for the fixup information is kept separate because the fixup data is not always loaded into main memory with the 'loader section'. If the checksum feature is not implemented, then the linker will set these fields to zero.

LOADER SECTION SIZE = DD Size of memory resident tables. This is the total size in bytes of the tables required to be memory resident for the module, while the module is in use. This total size includes all tables from the Object Table down to and including the Per-Page Checksum Table.

LOADER SECTION CHECKSUM = DD Checksum for loader section. This is a cryptographic checksum covering all of the loader section information. If the checksum feature is not implemented, then the linker will set these fields to zero.

OBJECT TABLE OFF = DD Object Table offset. This offset is relative to the beginning of the linear EXE header.

OBJECTS IN MODULE = DD Object Table Count. This defines the number of entries in Object Table.

OBJECT PAGE TABLE OFFSET = DD Object Page Table offset This offset is relative to the beginning of the linear EXE header.

OBJECT ITER PAGES OFF = DD Object Iterated Pages offset. This offset is relative to the beginning of the EXE file.

RESOURCE TABLE OFF = DD Resource Table offset. This offset is relative to the beginning of the linear EXE header.

RESOURCE TABLE ENTRIES = DD Number of entries in Resource Table.

RESIDENT NAME TBL OFF = DD Resident Name Table offset. This offset is relative to the beginning of the linear EXE header.

ENTRY TBL OFF = DD Entry Table offset. This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES OFF = DD Module Format Directives Table offset. This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES = DD Number of Module Format Directives in the Table. This field specifies the number of entries in the Module Format Directives Table.

FIXUP PAGE TABLE OFF = DD Fixup Page Table offset. This offset is relative to the beginning of the linear EXE header.

FIXUP RECORD TABLE OFF = DD Fixup Record Table Offset This offset is relative to the beginning of the linear EXE header.

IMPORT MODULE TBL OFF = DD Import Module Name Table offset. This offset is relative to the beginning of the linear EXE header.

IMPORT MOD ENTRIES = DD The number of entries in the Import Module Name Table.

IMPORT PROC TBL OFF = DD Import Procedure Name Table offset. This offset is relative to the beginning of the linear EXE header.

PER-PAGE CHECKSUM OFF = DD Per-Page Checksum Table offset. This offset is relative to the beginning of the linear EXE header.

DATA PAGES OFFSET = DD Data Pages Offset. This offset is relative to the beginning of the EXE file.

PRELOAD PAGES = DD Number of Preload pages for this module.

NON-RES NAME TBL OFF = DD Non-Resident Name Table offset. This offset is relative to the beginning of the EXE file.

NON-RES NAME TBL LEN = DD Number of bytes in the Non-resident name table.

NON-RES NAME TBL CKSM = DD Non-Resident Name Table Checksum. This is a cryptographic checksum of the Non-Resident Name Table.

AUTO DS OBJECT # = DD The Auto Data Segment Object number. This is the object number for the Auto Data Segment used by 16-bit modules. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

DEBUG INFO OFF = DD Debug Information offset. This offset is relative to the beginning of the linear EXE header.

DEBUG INFO LEN = DD Debug Information length. The length of the debug information in bytes.

INSTANCE PRELOAD = DD Instance pages in preload section. The number of instance data pages found in the preload section.

INSTANCE DEMAND = DD Instance pages in demand section. The number of instance data pages found in the demand section.

HEAPSIZE = DD Heap size added to the Auto DS Object. The heap size is the number of bytes added to the Auto Data Segment by the loader. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

Program (EXE) startup registers and Library entry registers

Program startup registers are defined as follows.

EIP = Starting program entry address.

ESP = Top of stack address.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to routine which calls DosExit(1,EAX).

[ESP+4] = Module handle for program module.

[ESP+8] = Reserved.

[ESP+12] = Environment data object address.

[ESP+16] = Command line linear address in environment data object.

Library initialization registers are defined as follows.

EIP = Library entry address.

ESP = User program stack.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

Note that a 32-bit Protected Memory Library module will be given a GDT selector in the DS and ES registers (PROTDS) that addresses the full linear address space available to a application. This selector should be saved by the initialization routine. Non-Protected Memory Library modules will receive a selector (FLATDS) that addresses the same amount of linear address space as an application's .EXE can.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to system, (EAX) = return code.

[ESP+4] = Module handle for library module.

[ESP+8] = 0 (Initialization)

Note that a 32-bit library may specify that its entry address is in a 16-bit code object. In this case, the entry registers are the same as for entry to a library using the Segmented EXE format. These are documented elsewhere. This means that a 16-bit library may be relinked to take advantage of the benefits of the Linear EXE format (notably, efficient paging).

Library termination registers are defined as follows.

EIP = Library entry address.

ESP = User program stack.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to system.

[ESP+4] = Module handle for library module.

[ESP+8] = 1 (Termination)

Note that Library termination is not allowed for libraries with 16-bit entries.

Object Table

The number of entries in the Object Table is given by the # Objects in Module field in the linear EXE header. Entries in the Object Table are

numbered starting from one.

Each Object Table entry has the following format:

00h	VIRTUAL SIZE	RELOC BASE ADDR
08h	OBJECT FLAGS	PAGE TABLE INDEX
10h	# PAGE TABLE ENTRIES	RESERVED

Object Table

VIRTUAL SIZE = DD Virtual memory size. This is the size of the object that will be allocated when the object is loaded. The object's virtual size (rounded up to the page size value) must be greater than or equal to the total size of the pages in the EXE file for the object. This memory size must also be large enough to contain all of the iterated data and uninitialized data in the EXE file.

RELOC BASE ADDR = DD Relocation Base Address. The relocation base address the object is currently relocated to. If the internal relocation fixups for the module have been removed, this is the address the object will be allocated at by the loader.

OBJECT FLAGS = DW Flag bits for the object. The object flag bits have the following definitions.

0001h = Readable Object.
0002h = Writable Object.
0004h = Executable Object. The readable, writable and executable flags provide support for all possible protections. In systems where all of these protections are not supported, the loader will be responsible for making the appropriate protection match for the system.
0008h = Resource Object.
0010h = Discardable Object.
0020h = Object is Shared.
0040h = Object has Preload Pages.
0080h = Object has Invalid Pages.
0100h = Object has Zero Filled Pages.
0200h = Object is Resident (valid for VDDs, PDDs only).
0300h = Object is Resident : Contiguous (VDDs, PDDs only).
0400h = Object is Resident : 'long-lockable' (VDDs, PDDs only).
0800h = Reserved for system use.
1000h = 16:16 Alias Required (80x86 Specific).
2000h = Big/Default Bit Setting (80x86 Specific). The 'big/default' bit , for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment descriptor. (The Default bit, or D-bit, determines whether the default word size is 32-bits or 16-bits. It also affects the interpretation of the instruction stream.)
4000h = Object is conforming for code (80x86 Specific).
8000h = Object I/O privilege level (80x86 Specific). Only used for 16:16 Alias Objects.

PAGE TABLE INDEX = DD Object Page Table Index. This specifies the number of the first object page table entry for this object. The object page table specifies where in the EXE file a page can be found for a given object and specifies per-page attributes.

The object table entries are ordered by logical page in the object table. In other words the object table entries are sorted based on the object page table index value.

PAGE TABLE ENTRIES = DD # of object page table entries for this object. Any logical pages at the end of an object that do not have an entry in the object page table associated with them are handled as zero filled or invalid pages by the loader.

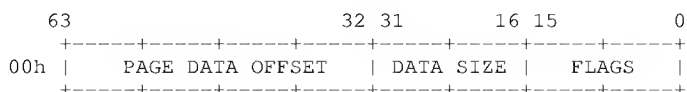
When the last logical pages of an object are not specified with an object page table entry, they are treated as either zero filled pages or invalid pages based on the last entry in the object page table for that object. If the last entry was neither a zero filled or invalid page, then the additional pages are treated as zero filled pages.

RESERVED = DD Reserved for future use. Must be set to zero.

Object Page Table

The Object page table provides information about a logical page in an object. A logical page may be an enumerated page, a pseudo page or an iterated page. The structure of the object page table in conjunction with the structure of the object table allows for efficient access of a page when a page fault occurs, while still allowing the physical page data to be located in the preload page, demand load page or iterated data page sections in the linear EXE module. The logical page entries in the Object Page Table are numbered starting from one. The Object Page Table is parallel to the Fixup Page Table as they are both indexed by the logical page number.

Each Object Page Table entry has the following format:



Object Page Table Entry

PAGE DATA OFFSET = DD Offset to the page data in the EXE file. This field, when bit shifted left by the PAGE OFFSET SHIFT from the module header, specifies the offset from the beginning of the Preload Page section of the physical page data in the EXE file that corresponds to this logical page entry. The page data may reside in the Preload Pages, Demand Load Pages or the Iterated Data Pages sections.

If the FLAGS field specifies that this is a zero-Filled page then the PAGE DATA OFFSET field will contain a 0.

If the logical page is specified as an iterated data page, as indicated by the FLAGS field, then this field specifies the offset into the Iterated Data Pages section.

The logical page number (Object Page Table index), is used to index the Fixup Page Table to find any fixups associated with the logical page.

DATA SIZE = DW Number of bytes of data for this page. This field specifies the actual number of bytes that represent the page in the file. If the PAGE SIZE field from the module header is greater than the value of this field and the FLAGS field indicates a Legal Physical Page, the remaining bytes are to be filled with zeros. If the FLAGS field indicates an Iterated Data Page, the iterated data records will completely fill out the remainder.

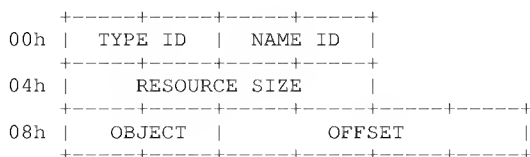
FLAGS = DW Attributes specifying characteristics of this logical page. The bit definitions for this word field follow,

- 00h = Legal Physical Page in the module (Offset from Preload Page Section).
- 01h = Iterated Data Page (Offset from Iterated Data Pages Section).
- 02h = Invalid Page (zero).
- 03h = Zero Filled Page (zero).
- 04h = Range of Pages.

Resource Table

The resource table is an array of resource table entries. Each resource table entry contains a type ID and name ID. These entries are used to locate resource objects contained in the Object table. The number of entries in the resource table is defined by the Resource Table Count located in the linear EXE header. More than one resource may be contained within a single object. Resource table entries are in a sorted order, (ascending, by Resource Name ID within the Resource Type ID). This allows the DosGetResource API function to use a binary search when looking up a resource in a 32-bit module instead of the linear search being used in the current 16-bit module.

Each resource entry has the following format:



Resource Table

TYPE ID = DW Resource type ID. The type of resources are:

- BTMP = Bitmap
- EMSG = Error message string
- FONT = Fonts

NAME ID = DW An ID used as a name for the resource when referred to.

RESOURCE SIZE = DD The number of bytes the resource consists of.

OBJECT = DW The number of the object which contains the resource.

OFFSET = DD The offset within the specified object where the resource begins.

Resident or Non-resident Name Table Entry

The resident and non-resident name tables define the ASCII names and ordinal numbers for exported entries in the module. In addition the first entry in the resident name table contains the module name. These tables are used to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the entry table.

The resident name table is kept resident in system memory while the module is loaded. It is intended to contain the exported entry point names that are frequently dynamically linked to by name. Non-resident names are not kept in memory and are read from the EXE file when a dynamic link reference is made. Exported entry point names that are infrequently dynamically linked to by name or are commonly referenced by ordinal number should be placed in the non-resident name table. The trade off made for references by name is performance vs memory usage.

Import references by name require these tables to be searched to obtain the entry point ordinal number. Import references by ordinal number provide the fastest lookup since the search of these tables is not required.

The strings are CASE SENSITIVE and are NOT NULL TERMINATED.

Each name table entry has the following format:

00h	+	-----	+	-----	+	-----	+	-----	+	+	-----	+	-----	+	-----	+
		LEN				ASCII		STRING		.	.	.			ORDINAL	#
	+	-----	+	-----	+	-----	+	-----	+	+	-----	+	-----	+	-----	+

Resident or Non-resident Name Table Entry

- LEN = DB String Length. This defines the length of the string in bytes. A zero length indicates there are no more entries in table. The length of each ascii name string is limited to 127 characters.
- The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.
- ASCII STRING = DB ASCII String. This is a variable length string with it's length defined in bytes by the LEN field. The string is case case sensitive and is not null terminated.
- ORDINAL # = DW Ordinal number. The ordinal number in an ordered index into the entry table for this entry point.

Entry Table

The entry table contains object and offset information that is used to resolve fixup references to the entry points within this module. Not all entry points in the entry table will be exported, some entry points will only be used within the module. An ordinal number is used to index into the entry table. The entry table entries are numbered starting from one.

The list of entries are compressed into 'bundles', where possible. The entries within each bundle are all the same size. A bundle starts with a count field which indicates the number of entries in the bundle. The count is followed by a type field which identifies the bundle format. This provides both a means for saving space as well as a mechanism for extending the bundle types.

The type field allows the definition of 256 bundle types. The following bundle types will initially be defined:

- Unused Entry.

16-bit Entry.
 286 Call Gate Entry.
 32-bit Entry.
 Forwarder Entry.

The bundled entry table has the following format:

```

+-----+-----+-----+-----+
00h | CNT | TYPE | BUNDLE INFO . . .
+-----+-----+-----+-----+
```

Entry Table

CNT = DB Number of entries. This is the number of entries in this bundle.

A zero value for the number of entries identifies the end of the entry table. There is no further bundle information when the number of entries is zero. In other words the entry table is terminated by a single zero byte.

TYPE = DB Bundle type. This defines the bundle type which determines the contents of the BUNDLE INFO.

The follow types are defined:

00h = Unused Entry.
 01h = 16-bit Entry.
 02h = 286 Call Gate Entry.
 03h = 32-bit Entry.
 04h = Forwarder Entry.
 80h = Parameter Typing Information Present. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

The following is the format for each bundle type:

```

+-----+-----+
00h | CNT | TYPE |
+-----+-----+
```

CNT = DB Number of entries. This is the number of unused entries to skip.

TYPE = DB 0 (Unused Entry)

```

+-----+-----+-----+-----+
00h | CNT | TYPE | OBJECT |
+-----+-----+-----+-----+
04h | FLAGS | OFFSET |
+-----+-----+-----+-----+
07h | ... | . . . |
+ + + +
```

CNT = DB Number of entries. This is the number of 16-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 1 (16-bit Entry)

OBJECT = DW Object number. This is the object number for the entries in this bundle.

FLAGS = DB Entry flags. These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
 F8h = Parameter word count mask.

OFFSET = DW Offset in object. This is the offset in the object for the entry point defined at this ordinal number.

00h	CNT	TYPE	OBJECT
04h	FLAGS	OFFSET	CALLGATE
09h

CNT = DB Number of entries. This is the number of 286 call gate entries in this bundle. The flags, callgate, and offset value are repeated this number of times.

TYPE = DB 2 (286 Call Gate Entry) The 286 Call Gate Entry Point type is needed by the loader only if ring 2 segments are to be supported. 286 Call Gate entries contain 2 extra bytes which are used by the loader to store an LDT callgate selector value.

OBJECT = DW Object number. This is the object number for the entries in this bundle.

FLAGS = DB Entry flags. These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object. This is the offset in the object for the entry point defined at this ordinal number.

CALLGATE = DW Callgate selector. The callgate selector is a reserved field used by the loader to store a call gate selector value for references to ring 2 entry points. When a ring 3 reference to a ring 2 entry point is made, the callgate selector with a zero offset is placed in the relocation fixup address. The segment number and offset in segment is placed in the LDT callgate.

00h	CNT	TYPE	OBJECT
04h	FLAGS	OFFSET	
09h	

CNT = DB Number of entries. This is the number of 32-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 3 (32-bit Entry) The 32-bit Entry type will only be defined by the linker when the offset in the object can not be specified by a 16-bit offset.

OBJECT = DW Object number. This is the object number for the entries in this bundle.

FLAGS = DB Entry flags. These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter dword count mask.

OFFSET = DD Offset in object. This is the offset in the object for the entry point defined at this ordinal number.

00h	CNT	TYPE	RESERVED
04h	FLAGS	MOD ORD#	OFFSET / ORDNUM
09h

CNT = DB Number of entries. This is the number of forwarder entries in this bundle. The FLAGS, MOD ORD#, and OFFSET/ORDNUM values are repeated this number of times.

TYPE = DB 4 (Forwarder Entry)

RESERVED = DW 0 This field is reserved for future use.

FLAGS = DB Forwarder flags. These are the flags for this entry point. They have the following definition.

01h = Import by ordinal.

F7h = Reserved for future use; should be zero.

MOD ORD# = DW Module Ordinal Number This is the index into the Import Module Name Table for this forwarder.

OFFSET / ORDNUM = DD Procedure Name Offset or Import Ordinal Number If the FLAGS field indicates import by ordinal, then this field is the ordinal number into the Entry Table of the target module, otherwise this field is the offset into the Procedure Names Table of the target module.

A Forwarder entry (type = 4) is an entry point whose value is an imported reference. When a load time fixup occurs whose target is a forwarder, the loader obtains the address imported by the forwarder and uses that imported address to resolve the fixup.

A forwarder may refer to an entry point in another module which is itself a forwarder, so there can be a chain of forwarders. The loader will traverse the chain until it finds a non-forwarded entry point which terminates the chain, and use this to resolve the original fixup. Circular chains are detected by the loader and result in a load time error. A maximum of 1024 forwarders is allowed in a chain; more than this results in a load time error.

Forwarders are useful for merging and recombining API calls into different sets of libraries, while maintaining compatibility with applications. For example, if one wanted to combine MONCALLS, MOUCALLS, and VIOCALLS into a single libraries, one could provide entry points for the three libraries that are forwarders pointing to the common implementation.

Module Format Directives Table

The Module Format Directives Table is an optional table that allows additional options to be specified. It also allows for the extension of the linear EXE format by allowing additional tables of information to be added to the linear EXE module without affecting the format of the linear EXE header. Likewise, module format directives provide a place in the linear EXE module for 'temporary tables' of information, such as incremental linking information and statistic information gathered on the module. When there are no module format directives for a linear EXE module, the fields in the linear EXE header referencing the module format directives table are zero.

Each Module Format Directive Table entry has the following format:

```
00h | DIRECT # | DATA LEN | DATA OFFSET |
+-----+-----+-----+-----+
```

Module Format Directive Table

DIRECT # = DW Directive number. The directive number specifies the type of directive defined. This can be used to determine the format of the information in the directive data. The following directive numbers have been defined:

8000h = Resident Flag Mask. Directive numbers with this bit set indicate that the directive data is in the resident area and will be kept resident in memory when the module is loaded.

8001h = Verify Record Directive. (Verify record is a resident table.)

0002h = Language Information Directive. (This is a non-resident table.)

0003h = Co-Processor Required Support Table.

0004h = Thread State Initialization Directive.

Additional directives can be added as needed in the future, as long as they do not overlap previously defined directive numbers.

DATA LEN = DW Directive data length. This specifies the length in byte of the directive data for this directive number.

DIRECTIVE OFFSET = DD Directive data offset. This is the offset to the directive data for this directive number. It is relative to beginning of linear EXE header for a resident table, and relative to the beginning of the EXE file for non-resident tables.

Verify Record Directive Table

The Verify Record Directive Table is an optional table. It maintains a record of the pages in the EXE file that have been fixed up and written

back to the original linear EXE module, along with the module dependencies used to perform these fixups. This table provides an efficient means for verifying the virtual addresses required for the fixed up pages when the module is loaded.

Each Verify Record entry has the following format:

00h	# OF ENTRY		
02h	MOD ORD #	VERSION	MOD # OBJ
08h	OBJECT #	BASE ADDR	VIRTUAL
0Eh	.	.	.

Verify Record Table

- # OF ENTRY = DW Number of module dependencies. This field specifies how many entries there are in the verify record directive table. This is equal to the number of modules referenced by this module.
- MOD ORD # = DW Ordinal index into the Import Module Name Table. This value is an ordered index in to the Import Module Name Table for the referenced module.
- VERSION = DW Module Version. This is the version of the referenced module that the fixups were originally performed. This is used to insure the same version of the referenced module is loaded that was fixed up in this module and therefore the fixups are still correct. This requires the version number in a module to be incremented anytime the entry point offsets change.
- MOD # OBJ = DW Module # of Object Entries. This field is used to identify the number of object verify entries that follow for the referenced module.
- OBJECT # = DW Object # in Module. This field specifies the object number in the referenced module that is being verified.
- BASE ADDR = DW Object load base address. This is the address that the object was loaded at when the fixups were performed.
- VIRTUAL = DW Object virtual address size. This field specifies the total amount of virtual memory required for this object.

Per-Page Checksum

The Per-Page Checksum table provides space for a cryptographic checksum for each physical page in the EXE file.

The checksum table is arranged such that the first entry in the table corresponds to the first logical page of code/data in the EXE file (usually a preload page) and the last entry corresponds to the last logical page in the EXE file (usually a iterated data page).

Logical Page #1	CHECKSUM
Logical Page #2	CHECKSUM
	.
Logical Page #n	CHECKSUM

Per-Page Checksum

CHECKSUM = DD Cryptographic checksum.

Fixup Page Table

The Fixup Page Table provides a simple mapping of a logical page number to an offset into the Fixup Record Table for that page.

This table is parallel to the Object Page Table, except that there is one additional entry in this table to indicate the end of the Fixup Record Table.

The format of each entry is:

Logical Page #1		OFFSET FOR PAGE #1		
Logical Page #2		OFFSET FOR PAGE #2		
		.	.	.
Logical Page #n		OFFSET FOR PAGE #n		
		OFF TO END OF FIXUP REC		
	This is equal to: Offset for page #n + Size of fixups for page #n			

Fixup Page Table

OFFSET FOR PAGE # = DD Offset for fixup record for this page. This field specifies the offset, from the beginning of the fixup record table, to the first fixup record for this page.

OFF TO END OF FIXUP REC = DD Offset to the end of the fixup records. This field specifies the offset following the last fixup record in the fixup record table. This is the last entry in the fixup page table.

The fixup records are kept in order by logical page in the fixup record table. This allows the end of each page's fixup records is defined by the offset for the next logical page's fixup records. This last entry provides support of this mechanism for the last page in the fixup page table.

Fixup Record Table

The Fixup Record Table contains entries for all fixups in the linear EXE module. The fixup records for a logical page are grouped together and kept in sorted order by logical page number. The fixups for each page are further sorted such that all external fixups and internal selector/pointer fixups come before internal non-selector/non-pointer fixups. This allows the loader to ignore internal fixups if the loader is able to load all objects at the addresses specified in the object table.

Each relocation record has the following format:

	+	-	-	-	+	+	-	-	-	+	+	-	-	+	+
00h		SRC		FLAGS		SRCOFF/CNT*									
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
03h/04h						TARGET DATA *									
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
		SRCOFF1 @		.		.		.		SRCOFFn @					
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

* These fields are variable size.

@ These fields are optional.

Fixup Record Table

SRC = DB Source type. The source type specifies the size and type of the fixup to be performed on the fixup source. The source type is defined as follows:

- 0Fh = Source mask.
- 00h = Byte fixup (8-bits).
- 01h = (undefined).
- 02h = 16-bit Selector fixup (16-bits).
- 03h = 16:16 Pointer fixup (32-bits).
- 04h = (undefined).
- 05h = 16-bit Offset fixup (16-bits).

06h = 16:32 Pointer fixup (48-bits).
 07h = 32-bit Offset fixup (32-bits).
 08h = 32-bit Self-relative offset fixup (32-bits).
 10h = Fixup to Alias Flag. When the 'Fixup to Alias' Flag is set, the source fixup refers to the 16:16 alias for the object. This is only valid for source types of 2, 3, and 6. For fixups such as this, the linker and loader will be required to perform additional checks such as ensuring that the target offset for this fixup is less than 64K.
 20h = Source List Flag. When the 'Source List' Flag is set, the SRCOFF field is compressed to a byte and contains the number of source offsets, and a list of source offsets follows the end of fixup record (after the optional additive value).

FLAGS = DB Target Flags. The target flags specify how the target information is interpreted. The target flags are defined as follows:

03h = Fixup target type mask.
 00h = Internal reference.
 01h = Imported reference by ordinal.
 02h = Imported reference by name.
 03h = Internal reference via entry table.
 04h = Additive Fixup Flag. When set, an additive value trails the fixup record (before the optional source offset list).
 08h = Reserved. Must be zero.
 10h = 32-bit Target Offset Flag. When set, the target offset is 32-bits, otherwise it is 16-bits.
 20h = 32-bit Additive Fixup Flag. When set, the additive value is 32-bits, otherwise it is 16-bits.
 40h = 16-bit Object Number/Module Ordinal Flag. When set, the object number or module ordinal number is 16-bits, otherwise it is 8-bits.
 80h = 8-bit Ordinal Flag. When set, the ordinal number is 8-bits, otherwise it is 16-bits.

SRCOFF = DW/CNT = DB Source offset or source offset list count. This field contains either an offset or a count depending on the Source List Flag. If the Source List Flag is set, a list of source offsets follows the additive field and this field contains the count of the entries in the source offset list. Otherwise, this is the single source offset for the fixup. Source offsets are relative to the beginning of the page where the fixup is to be made.

Note that for fixups that cross page boundaries, a separate fixup record is specified for each page. An offset is still used for the 2nd page but it now becomes a negative offset since the fixup originated on the preceding page. (For example, if only the last one byte of a 32-bit address is on the page to be fixed up, then the offset would have a value of -3.)

TARGET DATA = Target data for fixup. The format of the TARGET DATA is dependent upon target flags.

SRCOFF1 - SRCOFFn = DW[] Source offset list. This list is present if the Source List Flag is set in the Target Flags field. The number of entries in the source offset list is defined in the SRCOFF/CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.

```

      +-----+-----+-----+-----+
00h  | SRC  |FLAGS|SRCOFF/CNT*|
      +-----+-----+-----+-----+
03h/04h | OBJECT * |          TRGOFF * @          |
      +-----+-----+-----+-----+
      | SRCOFF1 @ | . . . | SRCOFFn @ |
      +-----+-----+-----+-----+

```

* These fields are variable size.
 @ These fields are optional.

Internal Fixup Record

OBJECT = D[B|W] Target object number. This field is an index into the current module's Object Table to specify the target Object. It is a Byte value when the '16-bit Object Number/Module Ordinal Flag' bit in the target flags field is clear and a Word value when the bit is set.

TRGOFF = D[W|D] Target offset. This field is an offset into the specified target Object. It is not present when the Source Type specifies a 16-bit Selector fixup. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

```

      +-----+-----+-----+-----+
00h  | SRC  |FLAGS|SRCOFF/CNT*|
      +-----+-----+-----+-----+
03h/04h | MOD ORD# * | PROCEDURE NAME OFFSET* | ADDITIVE * @ |
      +-----+-----+-----+-----+
      | SRCOFF1 @ | . . . | SRCOFFn @ |
      +-----+-----+-----+-----+

```

```

+-----+-----+-----+-----+

```

* These fields are variable size.
 @ These fields are optional.

Import by Name Fixup Record

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table. This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

PROCEDURE NAME OFFSET = D[W|D] Offset into the Import Procedure Name Table. This field is an offset into the Import Procedure Name Table. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value. This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

```

+-----+-----+-----+-----+
00h | SRC |FLAGS|SRCOFF/CNT*|
+-----+-----+-----+-----+
03h/04h | MOD ORD# *|IMPORT ORD*|      ADDITIVE * @      |
+-----+-----+-----+-----+
        | SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.
 @ These fields are optional.

Import by Ordinal Fixup Record

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table. This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

IMPORT ORD = D[B|W|D] Imported ordinal number. This is the imported procedure's ordinal number. It is a Byte value when the '8-bit Ordinal' bit in the target flags field is set. Otherwise it is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value. This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

```

+-----+-----+-----+-----+
00h | SRC |FLAGS|SRCOFF/CNT*|
+-----+-----+-----+-----+
03h/04h | ORD # * |      ADDITIVE * @      |
+-----+-----+-----+-----+
        | SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.
 @ These fields are optional.

Internal Entry Table Fixup Record

ENTRY # = D[B|W] Ordinal index into the Entry Table. This field is an index into the current module's Entry Table to specify the target Object and offset. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value. This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

Import Module Name Table

The import module name table defines the module name strings imported through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import module name table subtract the import module name table offset from the import procedure name table offset. These values are located in the linear EXE header. The end of the import module name table is not terminated by a special character, it is followed directly by the import procedure name table.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

```

+-----+-----+-----+-----+ +-----+
00h | LEN |   ASCII STRING   . . . |
+-----+-----+-----+-----+ +-----+
```

Import Module Name Table

LEN = DB String Length. This defines the length of the string in bytes. The length of each ascii name string is limited to 127 characters.

ASCII STRING = DB ASCII String. This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Import Procedure Name Table

The import procedure name table defines the procedure name strings imported by this module through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import procedure name table add the fixup section size to the fixup page table offset, this computes the offset to the end of the fixup section, then subtract the import procedure name table offset. These values are located in the linear EXE header. The import procedure name table is followed by the data pages section. Since the data pages section is aligned on a 'page size' boundary, padded space may exist between the last import name string and the first page in the data pages section. If this padded space exists it will be zero filled.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

```

+-----+-----+-----+-----+ +-----+
00h | LEN |   ASCII STRING   . . . |
+-----+-----+-----+-----+ +-----+
```

Import Procedure Name Table

LEN = DB String Length. This defines the length of the string in bytes. The length of each ascii name string is limited to 127 characters.

The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String. This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Preload Pages

The Preload Pages section is an optional section in the linear EXE module that coalesces a 'preload page set' into a contiguous section. The preload page set can be defined as the set of first used pages in the module. The preload page set can be specified by the application developer or can be derived by a tool that analyzes the programs memory usage while it is running. By grouping the preload page set together, the preload pages can be read from the linear EXE module with one disk read.

The structure of the preload pages is no different than if they were demand loaded. They are non-iterated pages. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded.

All pages begin on a PAGE OFFSET SHIFT boundary from the base of the preload page section, as specified in the linear EXE header. The pages are ordered by logical page number within this section.

Demand Load Pages

The Demand Loaded Pages section contains all the non-iterated pages for a linear EXE module that are not preloaded. When required, the whole page is loaded into memory from the module. The characteristics of each of these pages is specified in the Object Page Table. Every page begins on a PAGE OFFSET SHIFT boundary aligned offset from the demand loaded pages base specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded. The pages are ordered by logical page number within this section.

Iterated Data Pages

The Iterated Data Pages section contains all the pages for a linear EXE module that are iterated. When required, the set of iteration records are loaded into memory from the module and expanded to reconstitute the page. Every set of iteration records begins on a PAGE OFFSET SHIFT offset from the OBJECT ITER PAGES OFF specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. The pages are ordered by logical page number within this section.

This record structure is used to describe the iterated data for an object on a per-page basis.

```
+-----+-----+-----+-----+
00h |#ITERATIONS|DATA LENGTH|
+-----+-----+-----+-----+
04h |DATA BYTES | . . . | ... |
+-----+-----+-----+-----+
```

Object Iterated Data Record (Iteration Record)

#ITERATIONS = DW Number of iterations. This specifies the number of times that the data is replicated.

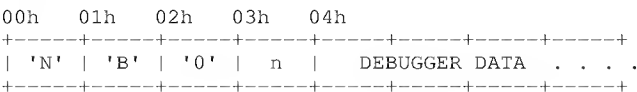
DATA LENGTH = DW The size of the data pattern in bytes. This specifies the number of bytes of data of which the pattern consists. The maximum size is one half of the PAGE SIZE (given in the module header). If a pattern exceeds this value then the data page will not be condensed into iterated data.

DATA = DB * DATA LENGTH The Data pattern to be replicated. The next iteration record will immediately follow the last byte of the pattern. The offset of the next iteration record is easily calculated from the offset of this record by adding the DATA LENGTH field and the sizes of the #ITERATIONS and DATA LENGTH fields.

Debug Information

The debug information is defined by the debugger and is not controlled by the linear EXE format or linker. The only data defined by the linear EXE format relative to the debug information is it's offset in the EXE file and length in bytes as defined in the linear EXE header.

To support multiple debuggers the first word of the debug information is a type field which determines the format of the debug information.



Debug Information

TYPE = DB DUP 4 Format type. This defines the type of debugger data that exists in the remainder of the debug information. The signature consists of a string of four (4) ASCII characters: "NB0" followed by the ASCII representation for 'n'. The values for 'n' are defined as follows.

These format types are defined.

- 00h = 32-bit CodeView debugger format.
- 01h = AIX debugger format.
- 02h = 16-bit CodeView debugger format.
- 04h = 32-bit OS/2 PM debugger (IBM) format.

DEBUGGER DATA = Debugger specific data. The format of the debugger data is defined by the debugger that is being used.

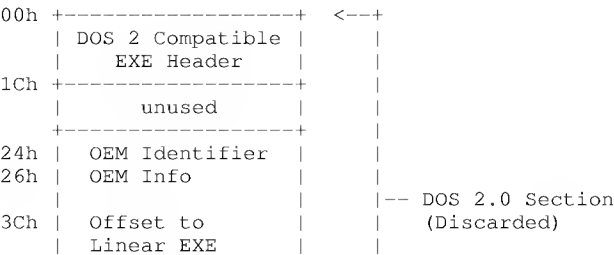
The values defined for the type field are not enforced by the system. It is the responsibility of the linker or debugging tools to follow the convention for the type field that is defined here.

LX specification - June 03, 1992

LX - Linear eXecutable Module Format Description

June 3, 1992

Figure 1. 32-bit Linear EXE File Layout



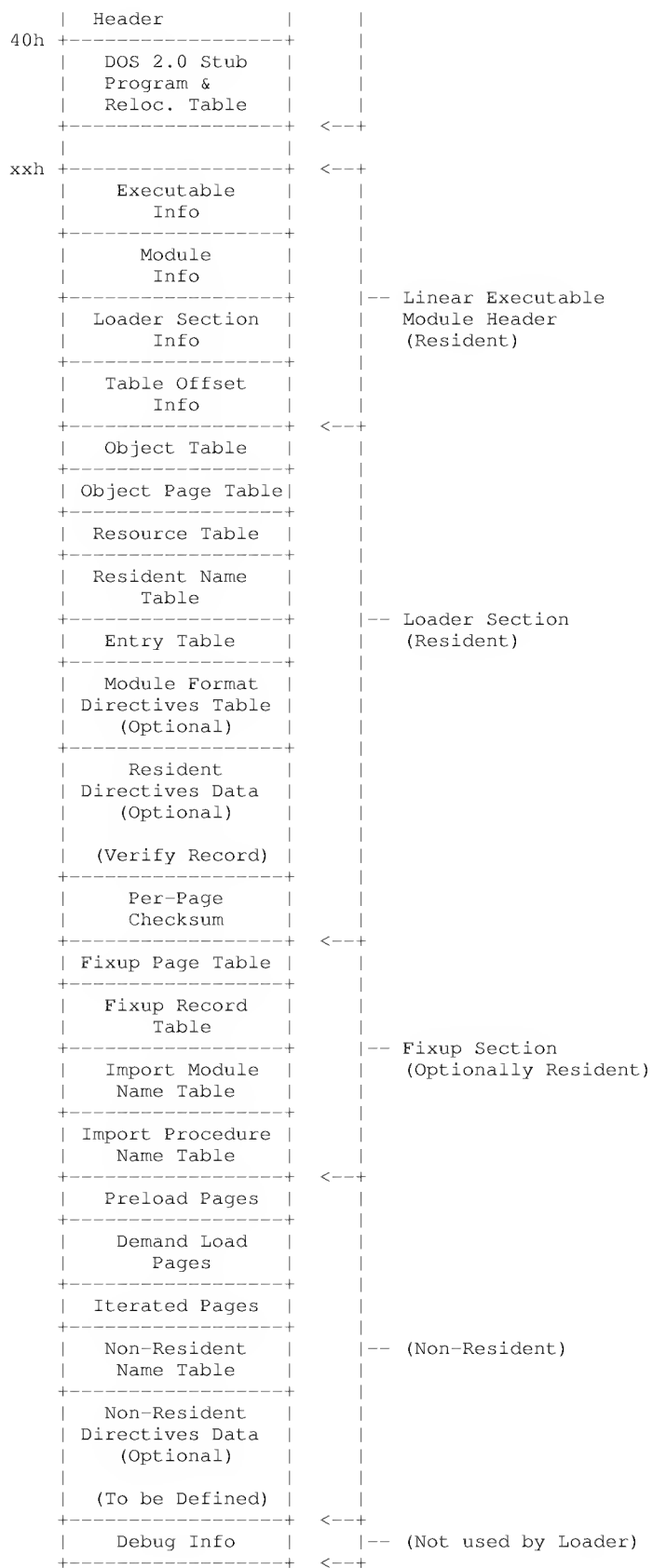
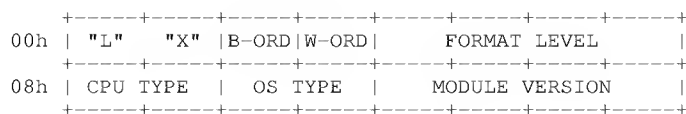


Figure 2. 32-bit Linear EXE Header



10h	MODULE FLAGS	MODULE # OF PAGES
18h	EIP OBJECT #	EIP
20h	ESP OBJECT #	ESP
28h	PAGE SIZE	PAGE OFFSET SHIFT
30h	FIXUP SECTION SIZE	FIXUP SECTION CHECKSUM
38h	LOADER SECTION SIZE	LOADER SECTION CHECKSUM
40h	OBJECT TABLE OFF	# OBJECTS IN MODULE
48h	OBJECT PAGE TABLE OFF	OBJECT ITER PAGES OFF
50h	RESOURCE TABLE OFFSET	#RESOURCE TABLE ENTRIES
58h	RESIDENT NAME TBL OFF	ENTRY TABLE OFFSET
60h	MODULE DIRECTIVES OFF	# MODULE DIRECTIVES
68h	FIXUP PAGE TABLE OFF	FIXUP RECORD TABLE OFF
70h	IMPORT MODULE TBL OFF	# IMPORT MOD ENTRIES
78h	IMPORT PROC TBL OFF	PER-PAGE CHECKSUM OFF
80h	DATA PAGES OFFSET	#PRELOAD PAGES
88h	NON-RES NAME TBL OFF	NON-RES NAME TBL LEN
90h	NON-RES NAME TBL CKSM	AUTO DS OBJECT #
98h	DEBUG INFO OFF	DEBUG INFO LEN
A0h	#INSTANCE PRELOAD	#INSTANCE DEMAND
A8h	HEAPSIZE	STACKSIZE

Note: The OBJECT ITER PAGES OFF must either be 0 or set to the same value as DATA PAGES OFFSET in OS/2 2.0. Ie., iterated pages are required to be in the same section of the file as regular pages.

Note: Table offsets in the Linear EXE Header may be set to zero to indicate that the table does not exist in the EXE file and it's size is zero.

"L" "X" = DW Signature word.

The signature word is used by the loader to identify the EXE file as a valid 32-bit Linear Executable Module Format. "L" is low order byte. "X" is high order byte.

B-ORD = DB Byte Ordering.

This byte specifies the byte ordering for the linear EXE format. The values are:

00H - Little Endian Byte Ordering.
01H - Big Endian Byte Ordering.

W-ORD = DB Word Ordering.

This byte specifies the Word ordering for the linear EXE format. The values are:

00H - Little Endian Word Ordering.
01H - Big Endian Word Ordering.

Format Level = DD Linear EXE Format Level.

The Linear EXE Format Level is set to 0 for the initial version of the 32-bit linear EXE format. Each incompatible change to the linear EXE format must increment this value. This allows the system to recognized future EXE file versions so that an appropriate error message may be displayed if an attempt is made to load them.

CPU Type = DW Module CPU Type.

This field specifies the type of CPU required by this module to run. The values are:

01H - 80286 or upwardly compatible CPU is

required to execute this module.
02H - 80386 or upwardly compatible CPU is
required to execute this module.
03H - 80486 or upwardly compatible CPU is
required to execute this module.

OS Type = DW Module OS Type.

This field specifies the type of Operating system
required to run this module. The currently defined
values are:

00H - Unknown (any "new-format" OS)
01H - OS/2 (default)
02H - Windows
03H - DOS 4.x
04H - Windows 386

MODULE VERSION = DD Version of the linear EXE module.

This is useful for differentiating between revisions
of dynamic linked modules. This value is specified
at link time by the user.

MODULE FLAGS = DD Flag bits for the module.

The module flag bits have the following definitions.

00000001h = Reserved for system use.
00000002h = Reserved for system use.
00000004h = Per-Process Library Initialization.
The setting of this bit requires the EIP
Object # and EIP fields to have valid
values. If the EIP Object # and EIP fields
are valid and this bit is NOT set, then
Global Library Initialization is assumed.
Setting this bit for an EXE file is invalid.

00000008h = Reserved for system use.
00000010h = Internal fixups for the module have
been applied.

The setting of this bit in a Linear
Executable Module indicates that each object
of the module has a preferred load address
specified in the Object Table Reloc Base
Addr. If the module's objects can not be
loaded at these preferred addresses, then
the relocation records that have been
retained in the file data will be applied.

00000020h = External fixups for the module have
been applied.

00000040h = Reserved for system use.
00000080h = Reserved for system use.
00000100h = Incompatible with PM windowing.
00000200h = Compatible with PM windowing.
00000300h = Uses PM windowing API.
00000400h = Reserved for system use.
00000800h = Reserved for system use.
00001000h = Reserved for system use.
00002000h = Module is not loadable.

When the 'Module is not loadable' flag is
set, it indicates that either errors were
detected at link time or that the module is
being incrementally linked and therefore
can't be loaded.

00004000h = Reserved for system use.

00038000h = Module type mask.

00000000h = Program module.

A module can not contain dynamic links to
other modules that have the 'program module'
type.

00008000h = Library module.

00018000h = Protected Memory Library module.

00020000h = Physical Device Driver module.

00028000h = Virtual Device Driver module.

40000000h = Per-process Library Termination.

The setting of this bit requires the EIP
Object # and EIP fields to have valid
values. If the EIP Object # and EIP fields
are valid and this bit is NOT set, then
Global Library Termination is assumed.
Setting this bit for an EXE file is invalid.

MODULE # PAGES = DD Number of pages in module.

This field specifies the number of pages physically contained in this module. In other words, pages containing either enumerated or iterated data, or zero-fill pages that have relocations, not invalid or zero-fill pages implied by the Virtual Size in the Object Table being larger than the number of pages actually in the linear EXE file. These pages are contained in the 'preload pages', 'demand load pages' and 'iterated data pages' sections of the linear EXE module. This is used to determine the size of the page information tables in the linear EXE module.

EIP OBJECT # = DD The Object number to which the Entry Address is relative.

This specifies the object to which the Entry Address is relative. This must be a nonzero value for a program module to be correctly loaded. A zero value for a library module indicates that no library entry routine exists. If this value is zero, then both the Per-process Library Initialization bit and the Per-process Library Termination bit must be clear in the module flags, or else the loader will fail to load the module. Further, if the Per-process Library Termination bit is set, then the object to which this field refers must be a 32-bit object (i.e., the Big/Default bit must be set in the object flags; see below).

EIP = DD Entry Address of module.

The Entry Address is the starting address for program modules and the library initialization and Library termination address for library modules.

ESP OBJECT # = DD The Object number to which the ESP is relative.

This specifies the object to which the starting ESP is relative. This must be a nonzero value for a program module to be correctly loaded. This field is ignored for a library module.

ESP = DD Starting stack address of module.

The ESP defines the starting stack pointer address for program modules. A zero value in this field indicates that the stack pointer is to be initialized to the highest address/offset in the object. This field is ignored for a library module.

PAGE SIZE = DD The size of one page for this system.

This field specifies the page size used by the linear EXE format and the system. For the initial version of this linear EXE format the page size is 4Kbytes. (The 4K page size is specified by a value of 4096 in this field.)

PAGE OFFSET SHIFT = DD The shift left bits for page offsets.

This field gives the number of bit positions to shift left when interpreting the Object Page Table entries' page offset field. This determines the alignment of the page information in the file. For example, a value of 4 in this field would align all pages in the Data Pages and Iterated Pages sections on 16 byte (paragraph) boundaries. A Page Offset Shift of 9 would align all pages on a 512 byte (disk sector) basis. The default value for this field is 12 (decimal), which give a 4096 byte alignment. All other offsets are byte aligned.

FIXUP SECTION SIZE = DD Total size of the fixup information in bytes.

This includes the following 4 tables:

- Fixup Page Table
- Fixup Record Table
- Import Module name Table
- Import Procedure Name Table

FIXUP SECTION CHECKSUM = DD Checksum for fixup information.

This is a cryptographic checksum covering all of the

fixup information. The checksum for the fixup information is kept separate because the fixup data is not always loaded into main memory with the 'loader section'. If the checksum feature is not implemented, then the linker will set these fields to zero.

LOADER SECTION SIZE = DD Size of memory resident tables.

This is the total size in bytes of the tables required to be memory resident for the module, while the module is in use. This total size includes all tables from the Object Table down to and including the Per-Page Checksum Table.

LOADER SECTION CHECKSUM = DD Checksum for loader section.

This is a cryptographic checksum covering all of the loader section information. If the checksum feature is not implemented, then the linker will set these fields to zero.

OBJECT TABLE OFF = DD Object Table offset.

This offset is relative to the beginning of the linear EXE header.

OBJECTS IN MODULE = DD Object Table Count.

This defines the number of entries in Object Table.

OBJECT PAGE TABLE OFFSET = DD Object Page Table offset

This offset is relative to the beginning of the linear EXE header.

OBJECT ITER PAGES OFF = DD Object Iterated Pages offset.

This offset is relative to the beginning of the EXE file.

RESOURCE TABLE OFF = DD Resource Table offset.

This offset is relative to the beginning of the linear EXE header.

RESOURCE TABLE ENTRIES = DD Number of entries in Resource Table.

RESIDENT NAME TBL OFF = DD Resident Name Table offset.

This offset is relative to the beginning of the linear EXE header.

ENTRY TBL OFF = DD Entry Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES OFF = DD Module Format Directives Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES = DD Number of Module Format Directives in the Table.

This field specifies the number of entries in the Module Format Directives Table.

FIXUP PAGE TABLE OFF = DD Fixup Page Table offset.

This offset is relative to the beginning of the linear EXE header.

FIXUP RECORD TABLE OFF = DD Fixup Record Table Offset

This offset is relative to the beginning of the linear EXE header.

IMPORT MODULE TBL OFF = DD Import Module Name Table offset.

This offset is relative to the beginning of the linear EXE header.

IMPORT MOD ENTRIES = DD The number of entries in the Import Module Name Table.

IMPORT PROC TBL OFF = DD Import Procedure Name Table offset.

This offset is relative to the beginning of the linear EXE header.

PER-PAGE CHECKSUM OFF = DD Per-Page Checksum Table offset.
 This offset is relative to the beginning of the linear EXE header.

DATA PAGES OFFSET = DD Data Pages Offset.
 This offset is relative to the beginning of the EXE file.

PRELOAD PAGES = DD Number of Preload pages for this module. Note that OS/2 2.0 does not respect the preload of pages as specified in the executable file for performance reasons.

NON-RES NAME TBL OFF = DD Non-Resident Name Table offset.
 This offset is relative to the beginning of the EXE file.

NON-RES NAME TBL LEN = DD Number of bytes in the Non-resident name table.

NON-RES NAME TBL CKSM = DD Non-Resident Name Table Checksum.
 This is a cryptographic checksum of the Non-Resident Name Table.

AUTO DS OBJECT # = DD The Auto Data Segment Object number.
 This is the object number for the Auto Data Segment used by 16-bit modules. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

DEBUG INFO OFF = DD Debug Information offset.
 This offset is relative to the beginning of the file.

Note, earlier versions of this doc stated that this offset was from the linear EXE header - this is incorrect.

DEBUG INFO LEN = DD Debug Information length.
 The length of the debug information in bytes.

INSTANCE PRELOAD = DD Instance pages in preload section.
 The number of instance data pages found in the preload section.

INSTANCE DEMAND = DD Instance pages in demand section.
 The number of instance data pages found in the demand section.

HEAPSIZE = DD Heap size added to the Auto DS Object.
 The heap size is the number of bytes added to the Auto Data Segment by the loader. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

STACKSIZE = DD Stack size used by the module. May be set to zero.

Startup/termination conditions

Program (EXE) startup registers and Library entry registers

Program startup registers are defined as follows.

EIP = Starting program entry address.
 ESP = Top of stack address.
 CS = Code selector for base of linear address space.
 DS = ES = SS = Data selector for base of linear address space.
 FS = Data selector of base of Thread Information Block (TIB).
 GS = 0.
 EAX = EBX = 0.
 ECX = EDX = 0.
 ESI = EDI = 0.
 EBP = 0.
 [ESP+0] = Return address to routine which calls DosExit(1,EAX).
 [ESP+4] = Module handle for program module.
 [ESP+8] = Reserved.
 [ESP+12] = Environment data object address.
 [ESP+16] = Command line linear address in environment data object.

Library initialization registers are defined as follows.

EIP = Library entry address.
 ESP = User program stack.
 CS = Code selector for base of linear address space.
 DS = ES = SS = Data selector for base of linear address space.
 Note that a 32-bit Protected Memory Library module will be given a GDT selector in the DS and ES registers (PROTDS) that addresses the full linear address space available to a application. This selector should be saved by the initialization routine. Non-Protected Memory Library modules will receive a selector (FLATDS) that addresses the same amount of linear address space as an application's .EXE can.
 FS = Data selector of base of Thread Information Block (TIB).
 GS = 0.
 EAX = EBX = 0.
 ECX = EDX = 0.
 ESI = EDI = 0.
 EBP = 0.
 [ESP+0] = Return address to system, (EAX) = return code.
 [ESP+4] = Module handle for library module.
 [ESP+8] = 0 (Initialization)

Note that a 32-bit library may specify that its entry address is in a 16-bit code object. In this case, the entry registers are the same as for entry to a library using the Segmented EXE format. These are documented elsewhere. This means that a 16-bit library may be relinked to take advantage of the benefits of the Linear EXE format (notably, efficient paging).

Library termination registers are defined as follows.

```
EIP = Library entry address.

ESP = User program stack.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear
address space.

FS = Data selector of base of Thread Information
Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to system.

[ESP+4] = Module handle for library module.

[ESP+8] = 1 (Termination)
```

Note that Library termination is not allowed for libraries with 16-bit entries.

Object Table

Object Table

The number of entries in the Object Table is given by the # Objects in Module field in the linear EXE header. Entries in the Object Table are numbered starting from one.

Each Object Table entry has the following format:

00h		VIRTUAL SIZE		RELOC BASE ADDR	
08h		OBJECT FLAGS		PAGE TABLE INDEX	
10h		# PAGE TABLE ENTRIES		RESERVED	

VIRTUAL SIZE = DD Virtual memory size.

This is the size of the object that will be allocated when the object is loaded. The object's virtual size (rounded up to the page size value) must be greater than or equal to the total size of the pages in the EXE file for the object. This memory size must also be large enough to contain all of the iterated data and uninitialized data in the EXE file.

RELOC BASE ADDR = DD Relocation Base Address.

The relocation base address the object is currently relocated to. If the internal relocation fixups for the module have been removed, this is the address the object will be allocated at by the loader.

OBJECT FLAGS = DW Flag bits for the object.
The object flag bits have the following definitions.

0001h = Readable Object.
0002h = Writable Object.
0004h = Executable Object.
The readable, writable and executable flags provide support for all possible protections. In systems where all of these protections are not supported, the loader will be responsible for making the appropriate protection match for the system.

0008h = Resource Object.
0010h = Discardable Object.
0020h = Object is Shared.
0040h = Object has Preload Pages.
0080h = Object has Invalid Pages.
0100h = Object has Zero Filled Pages.
0200h = Object is Resident (valid for VDDs, PDDs only).
0300h = Object is Resident & Contiguous (VDDs, PDDs only).
0400h = Object is Resident & 'long-lockable' (VDDs, PDDs only).
0800h = Reserved for system use.
1000h = 16:16 Alias Required (80x86 Specific).
2000h = Big/Default Bit Setting (80x86 Specific).

The 'big/default' bit, for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment descriptor. (The Default bit, or D-bit, determines whether the default word size is 32-bits or 16-bits. It also affects the interpretation of the instruction stream.)

4000h = Object is conforming for code (80x86 Specific).
8000h = Object I/O privilege level (80x86 Specific).
Only used for 16:16 Alias Objects.

PAGE TABLE INDEX = DD Object Page Table Index.

This specifies the number of the first object page table entry for this object. The object page table specifies where in the EXE file a page can be found for a given object and specifies per-page attributes.

The object table entries are ordered by logical page in the object table. In other words the object table entries are sorted based on the object page table index value.

PAGE TABLE ENTRIES = DD # of object page table entries for this object.

Any logical pages at the end of an object that do not have an entry in the object page table associated with them are handled as zero filled or invalid pages by the loader.

When the last logical pages of an object are not specified with an object page table entry, they are treated as either zero filled pages or invalid pages based on the last entry in the object page table for that object. If the last entry was neither a zero filled or invalid page, then the additional pages are treated as zero filled pages.

RESERVED = DD Reserved for future use. Must be set to zero.

Object Page Table

Object Page Table

The Object page table provides information about a logical page in an object. A logical page may be an enumerated page, a pseudo page or an iterated page. The structure of the object page table in conjunction with the structure of the object table allows for efficient access of a page when a page fault occurs, while still allowing the physical page data to be located in the preload page, demand load page or iterated data page sections in the linear EXE module. The logical page entries in the Object Page Table are numbered starting from one. The Object Page Table is parallel to the Fixup Page Table as they are both indexed by the logical page number.

Each Object Page Table entry has the following format:

	63		32 31		16 15		0
	+	-----	+	-----	+	-----	+
00h		PAGE DATA OFFSET		DATA SIZE		FLAGS	
	+	-----	+	-----	+	-----	+

PAGE DATA OFFSET = DD Offset to the page data in the EXE file.

This field, when bit shifted left by the PAGE OFFSET SHIFT from the module header, specifies the offset from the beginning of the Preload Page section of the physical page data in the EXE file that corresponds to this logical page entry. The page data may reside in the Preload Pages, Demand Load Pages or the Iterated Data Pages sections.

If the FLAGS field specifies that this is a zero-Filled page then the PAGE DATA OFFSET field will contain a 0.

If the logical page is specified as an iterated data page, as indicated by the FLAGS field, then this field specifies the offset into the Iterated Data Pages section.

The logical page number (Object Page Table index), is used to index the Fixup Page Table to find any fixups associated with the logical page.

DATA SIZE = DW Number of bytes of data for this page.

This field specifies the actual number of bytes that represent the page in the file. If the PAGE SIZE field from the module header is greater than the value of this field and the FLAGS field indicates a Legal Physical Page, the remaining bytes are to be filled with zeros. If the FLAGS field indicates an Iterated Data Page, the iterated data records will completely fill out the remainder.

FLAGS = DW Attributes specifying characteristics of this logical page.

The bit definitions for this word field follow,

- 00h = Legal Physical Page in the module (Offset from Preload Page Section).
- 01h = Iterated Data Page (Offset from Iterated Data Pages Section).
- 02h = Invalid Page (zero).
- 03h = Zero Filled Page (zero).
- 04h = Range of Pages.

Resource Table

Resource Table

The resource table is an array of resource table entries. Each resource table entry contains a type ID and name ID. These entries are used to locate resource objects contained in the Object table. The number of entries in the resource table is defined by the Resource Table Count located in the linear EXE header. More than one resource may be contained within a single object. Resource table entries are in a sorted order, (ascending, by Resource Name ID within the Resource Type ID). This allows the DosGetResource API function to use a binary search when looking up a resource in a 32-bit module instead of the linear search being used in the current 16-bit module.

Each resource entry has the following format:

```
+-----+-----+-----+-----+
00h | TYPE ID | NAME ID |
+-----+-----+-----+-----+
04h | RESOURCE SIZE |
+-----+-----+-----+-----+
08h | OBJECT | OFFSET |
+-----+-----+-----+-----+
```

TYPE ID = DW Resource type ID.
The type of resources are:

BTMP = Bitmap
EMSG = Error message string
FONT = Fonts

NAME ID = DW An ID used as a name for the resource when referred to.

RESOURCE SIZE = DD The number of bytes the resource consists of.

OBJECT = DW The number of the object which contains the resource.

OFFSET = DD The offset within the specified object where the resource begins.

Name Table Entries

Resident or Non-resident Name Table Entry

The resident and non-resident name tables define the ASCII names and ordinal numbers for exported entries in the module. In addition the first entry in the resident name table contains the module name. These tables are used to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the entry

table.

The resident name table is kept resident in system memory while the module is loaded. It is intended to contain the exported entry point names that are frequently dynamically linked to by name. Non-resident names are not kept in memory and are read from the EXE file when a dynamic link reference is made. Exported entry point names that are infrequently dynamically linked to by name or are commonly referenced by ordinal number should be placed in the non-resident name table. The trade off made for references by name is performance vs memory usage.

Import references by name require these tables to be searched to obtain the entry point ordinal number. Import references by ordinal number provide the fastest lookup since the search of these tables is not required.

The strings are CASE SENSITIVE and are NOT NULL TERMINATED.

Each name table entry has the following format:

```

+-----+-----+-----+-----+   +-----+-----+-----+
00h | LEN |   ASCII STRING   . . .   | ORDINAL # |
+-----+-----+-----+-----+   +-----+-----+-----+
```

LEN = DB String Length.

This defines the length of the string in bytes. A zero length indicates there are no more entries in table. The length of each ascii name string is limited to 127 characters.

The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case case sensitive and is not null terminated.

ORDINAL # = DW Ordinal number.

The ordinal number in an ordered index into the entry table for this entry point.

Entry Table

Entry Table

The entry table contains object and offset information that is used to resolve fixup references to the entry points within this module. Not all entry points in the entry table will be exported, some entry points will only be used within the module. An ordinal number is used to index into the entry table. The entry table entries are numbered starting from one.

The list of entries are compressed into 'bundles', where possible. The entries within each bundle are all the same size. A bundle starts with a count field which indicates the number of entries in the bundle. The count is followed by a type field which identifies the bundle format. This provides both a means for saving space as well as a mechanism for extending the bundle types.

The type field allows the definition of 256 bundle types.
The following bundle types will initially be defined:

Unused Entry.
16-bit Entry.
286 Call Gate Entry.
32-bit Entry.
Forwarder Entry.

The bundled entry table has the following format:

```

+-----+-----+-----+-----+
00h | CNT | TYPE | BUNDLE INFO . . .
+-----+-----+-----+-----+

```

CNT = DB Number of entries.
This is the number of entries in this bundle.

A zero value for the number of entries identifies the end of the entry table. There is no further bundle information when the number of entries is zero. In other words the entry table is terminated by a single zero byte.

TYPE = DB Bundle type.
This defines the bundle type which determines the contents of the BUNDLE INFO.

The follow types are defined:

00h = Unused Entry.
01h = 16-bit Entry.
02h = 286 Call Gate Entry.
03h = 32-bit Entry.
04h = Forwarder Entry.
80h = Parameter Typing Information Present.
This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

The following is the format for each bundle type:

```

+-----+-----+
00h | CNT | TYPE |
+-----+-----+

```

CNT = DB Number of entries.
This is the number of unused entries to skip.

TYPE = DB 0 (Unused Entry)

```

+-----+-----+-----+-----+
00h | CNT | TYPE | OBJECT |
+-----+-----+-----+-----+
04h | FLAGS | OFFSET |
+-----+-----+-----+
07h | ... | . . . |
+ + + +

```

CNT = DB Number of entries.
This is the number of 16-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 1 (16-bit Entry)

OBJECT = DW Object number.
This is the object number for the entries in this bundle.

FLAGS = DB Entry flags.
These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.
This is the offset in the object for the

entry point defined at this ordinal number.

	+-----+-----+-----+-----+
00h	CNT TYPE OBJECT
	+-----+-----+-----+-----+
04h	FLAGS OFFSET CALLGATE
	+-----+-----+-----+-----+
09h
	+ + + + +

CNT = DB Number of entries.
This is the number of 286 call gate entries in this bundle. The flags, callgate, and offset value are repeated this number of times.

TYPE = DB 2 (286 Call Gate Entry)
The 286 Call Gate Entry Point type is needed by the loader only if ring 2 segments are to be supported. 286 Call Gate entries contain 2 extra bytes which are used by the loader to store an LDT callgate selector value.

OBJECT = DW Object number.
This is the object number for the entries in this bundle.

FLAGS = DB Entry flags.
These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.
This is the offset in the object for the entry point defined at this ordinal number.

CALLGATE = DW Callgate selector.
The callgate selector is a reserved field used by the loader to store a call gate selector value for references to ring 2 entry points. When a ring 3 reference to a ring 2 entry point is made, the callgate selector with a zero offset is place in the relocation fixup address. The segment number and offset in segment is placed in the LDT callgate.

	+-----+-----+-----+-----+
00h	CNT TYPE OBJECT
	+-----+-----+-----+-----+
04h	FLAGS OFFSET
	+-----+-----+-----+-----+
09h
	+ + + + +

CNT = DB Number of entries.
This is the number of 32-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 3 (32-bit Entry)
The 32-bit Entry type will only be defined by the linker when the offset in the object can not be specified by a 16-bit offset.

OBJECT = DW Object number.
This is the object number for the entries in this bundle.

FLAGS = DB Entry flags.
These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter dword count mask.

OFFSET = DD Offset in object.
This is the offset in the object for the entry point defined at this ordinal number.

	+-----+-----+-----+-----+	
00h	CNT TYPE RESERVED	
	+-----+-----+-----+-----+	
04h	FLAGS MOD ORD# OFFSET / ORDNUM	
	+-----+-----+-----+-----+	
09h	
	+ + + + + + + +	

CNT = DB Number of entries.
This is the number of forwarder entries in this bundle. The FLAGS, MOD ORD#, and OFFSET/ORDNUM values are repeated this number of times.

TYPE = DB 4 (Forwarder Entry)

RESERVED = DW 0
This field is reserved for future use.

FLAGS = DB Forwarder flags.
These are the flags for this entry point. They have the following definition.

01h = Import by ordinal.
F7h = Reserved for future use; should be zero.

MOD ORD# = DW Module Ordinal Number
This is the index into the Import Module Name Table for this forwarder.

OFFSET / ORDNUM = DD Procedure Name Offset or Import Ordinal Number
If the FLAGS field indicates import by ordinal, then this field is the ordinal number into the Entry Table of the target module, otherwise this field is the offset into the Procedure Names Table of the target module.

A Forwarder entry (type = 4) is an entry point whose value is an imported reference. When a load time fixup occurs whose target is a forwarder, the loader obtains the address imported by the forwarder and uses that imported address to resolve the fixup.

A forwarder may refer to an entry point in another module which is itself a forwarder, so there can be a chain of forwarders. The loader will traverse the chain until it finds a non-forwarded entry point which terminates the chain, and use this to resolve the original fixup. Circular chains are detected by the loader and result in a load time error. A maximum of 1024 forwarders is allowed in a chain; more than this results in a load time error.

Forwarders are useful for merging and recombining API calls into different sets of libraries, while maintaining compatibility with applications. For example, if one wanted to combine MONCALLS, MOUCALLS, and VIOCALLS into a single libraries, one could provide entry points for the three libraries that are forwarders pointing to the common implementation.

Module Format Directives Table

Module Format Directives Table

The Module Format Directives Table is an optional table that allows additional options to be specified. It also allows for the extension of the linear EXE format by allowing additional tables of information to be added to the linear EXE module without affecting the format of the linear EXE

header. Likewise, module format directives provide a place in the linear EXE module for 'temporary tables' of information, such as incremental linking information and statistic information gathered on the module. When there are no module format directives for a linear EXE module, the fields in the linear EXE header referencing the module format directives table are zero.

Each Module Format Directive Table entry has the following format:

```

+-----+-----+-----+-----+-----+-----+-----+
00h | DIRECT # | DATA LEN | DATA OFFSET |
+-----+-----+-----+-----+-----+-----+-----+

```

DIRECT # = DW Directive number.

The directive number specifies the type of directive defined. This can be used to determine the format of the information in the directive data. The following directive numbers have been defined:

8000h = Resident Flag Mask.

Directive numbers with this bit set indicate that the directive data is in the resident area and will be kept resident in memory when the module is loaded.

8001h = Verify Record Directive. (Verify record is a resident table.)

0002h = Language Information Directive. (This is a non-resident table.)

0003h = Co-Processor Required Support Table.

0004h = Thread State Initialization Directive.

Additional directives can be added as needed in the future, as long as they do not overlap previously defined directive numbers.

DATA LEN = DW Directive data length.

This specifies the length in byte of the directive data for this directive number.

DIRECTIVE OFFSET = DD Directive data offset.

This is the offset to the directive data for this directive number. It is relative to beginning of linear EXE header for a resident table, and relative to the beginning of the EXE file for non-resident tables.

Verify Record Directive Table

Verify Record Directive Table

The Verify Record Directive Table is an optional table. It maintains a record of the pages in the EXE file that have been fixed up and written back to the original linear EXE module, along with the module dependencies used to perform these fixups. This table provides an efficient means for verifying the virtual addresses required for the fixed up pages when the module is loaded.

Each Verify Record entry has the following format:

```

+-----+-----+
00h | # OF ENTRY |
+-----+-----+-----+-----+
02h | MOD ORD # | VERSION | MOD # OBJ |

```

	+-----+-----+-----+-----+-----+
08h	OBJECT # BASE ADDR VIRTUAL
	+-----+-----+-----+-----+-----+
0Eh
	+ + + + + + + +

OF ENTRY = DW Number of module dependencies.
 This field specifies how many entries there are in the verify record directive table. This is equal to the number of modules referenced by this module.

MOD ORD # = DW Ordinal index into the Import Module Name Table.
 This value is an ordered index in to the Import Module Name Table for the referenced module.

VERSION = DW Module Version.
 This is the version of the referenced module that the fixups were originally performed. This is used to insure the same version of the referenced module is loaded that was fixed up in this module and therefore the fixups are still correct. This requires the version number in a module to be incremented anytime the entry point offsets change.

MOD # OBJ = DW Module # of Object Entries.
 This field is used to identify the number of object verify entries that follow for the referenced module.

OBJECT # = DW Object # in Module.
 This field specifies the object number in the referenced module that is being verified.

BASE ADDR = DW Object load base address.
 This is the address that the object was loaded at when the fixups were performed.

VIRTUAL = DW Object virtual address size.
 This field specifies the total amount of virtual memory required for this object.

Per-Page Checksum Table

Per-Page Checksum

The Per-Page Checksum table provides space for a cryptographic checksum for each physical page in the EXE file.

The checksum table is arranged such that the first entry in the table corresponds to the first logical page of code/data in the EXE file (usually a preload page) and the last entry corresponds to the last logical page in the EXE file (usually a iterated data page).

	+-----+-----+-----+-----+
Logical Page #1	CHECKSUM
	+-----+-----+-----+-----+
Logical Page #2	CHECKSUM
	+-----+-----+-----+-----+
	. . .
	+-----+-----+-----+-----+
Logical Page #n	CHECKSUM
	+-----+-----+-----+-----+

CHECKSUM = DD Cryptographic checksum.

Fixup Page Table

Fixup Page Table

The Fixup Page Table provides a simple mapping of a logical page number to an offset into the Fixup Record Table for that page.

This table is parallel to the Object Page Table, except that there is one additional entry in this table to indicate the end of the Fixup Record Table.

The format of each entry is:

Logical Page #1		OFFSET FOR PAGE #1		
Logical Page #2		OFFSET FOR PAGE #2		
		.	.	.
Logical Page #n		OFFSET FOR PAGE #n		
		OFF TO END OF FIXUP REC		This is equal to:
				Offset for page #n + Size
				of fixups for page #n

OFFSET FOR PAGE # = DD Offset for fixup record for this page.

This field specifies the offset, from the beginning of the fixup record table, to the first fixup record for this page.

OFF TO END OF FIXUP REC = DD Offset to the end of the fixup records.

This field specifies the offset following the last fixup record in the fixup record table. This is the last entry in the fixup page table.

The fixup records are kept in order by logical page in the fixup record table. This allows the end of each page's fixup records is defined by the offset for the next logical page's fixup records. This last entry provides support of this mechanism for the last page in the fixup page table.

Fixup Record Table

Fixup Record Table

The Fixup Record Table contains entries for all fixups in

the linear EXE module. The fixup records for a logical page are grouped together and kept in sorted order by logical page number. The fixups for each page are further sorted such that all external fixups and internal selector/pointer fixups come before internal non-selector/non-pointer fixups. This allows the loader to ignore internal fixups if the loader is able to load all objects at the addresses specified in the object table.

Each relocation record has the following format:

```

+-----+-----+-----+-----+
00h | SRC | FLAGS | SRCOFF/CNT* |
+-----+-----+-----+-----+
03h/04h |          TARGET DATA *          |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.

@ These fields are optional.

SRC = DB Source type.

The source type specifies the size and type of the fixup to be performed on the fixup source. The source type is defined as follows:

0Fh = Source mask.
00h = Byte fixup (8-bits).
01h = (undefined).
02h = 16-bit Selector fixup (16-bits).
03h = 16:16 Pointer fixup (32-bits).
04h = (undefined).
05h = 16-bit Offset fixup (16-bits).
06h = 16:32 Pointer fixup (48-bits).
07h = 32-bit Offset fixup (32-bits).
08h = 32-bit Self-relative offset fixup (32-bits).
10h = Fixup to Alias Flag.

When the 'Fixup to Alias' Flag is set, the source fixup refers to the 16:16 alias for the object. This is only valid for source types of 2, 3, and 6. For fixups such as this, the linker and loader will be required to perform additional checks such as ensuring that the target offset for this fixup is less than 64K.

20h = Source List Flag.

When the 'Source List' Flag is set, the SRCOFF field is compressed to a byte and contains the number of source offsets, and a list of source offsets follows the end of fixup record (after the optional additive value).

FLAGS = DB Target Flags.

The target flags specify how the target information is interpreted. The target flags are defined as follows:

03h = Fixup target type mask.
00h = Internal reference.
01h = Imported reference by ordinal.
02h = Imported reference by name.
03h = Internal reference via entry table.
04h = Additive Fixup Flag.
When set, an additive value trails the fixup record (before the optional source offset list).

08h = Reserved. Must be zero.

10h = 32-bit Target Offset Flag.

When set, the target offset is 32-bits, otherwise it is 16-bits.

20h = 32-bit Additive Fixup Flag.

When set, the additive value is 32-bits, otherwise it is 16-bits.

40h = 16-bit Object Number/Module Ordinal Flag.

When set, the object number or module ordinal number is 16-bits, otherwise it is 8-bits.

80h = 8-bit Ordinal Flag.

When set, the ordinal number is 8-bits, otherwise it is 16-bits.

SRCOFF = DW/CNT = DB Source offset or source offset list count.

This field contains either an offset or a count depending on the Source List Flag. If the Source List Flag is set, a list of source offsets follows the additive field and this field contains the count of the entries in the source offset list. Otherwise, this is the single source offset for the fixup. Source offsets are relative to the beginning of the page where the fixup is to be made.

Note that for fixups that cross page boundaries, a separate fixup record is specified for each page. An offset is still used for the 2nd page but it now becomes a negative offset since the fixup originated on the preceding page. (For example, if only the last one byte of a 32-bit address is on the page to be fixed up, then the offset would have a value of -3.)

TARGET DATA = Target data for fixup.

The format of the TARGET DATA is dependent upon target flags.

SRCOFF1 - SRCOFFn = DW[] Source offset list.

This list is present if the Source List Flag is set in the Target Flags field. The number of entries in the source offset list is defined in the SRCOFF/CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.

```

+-----+-----+-----+-----+
00h | SRC |FLAGS|SRCOFF/CNT*|
+-----+-----+-----+-----+
03h/04h | OBJECT * |          TRGOFF * @          |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.

@ These fields are optional.

OBJECT = D[B|W] Target object number.

This field is an index into the current module's Object Table to specify the target Object. It is a Byte value when the '16-bit Object Number/Module Ordinal Flag' bit in the target flags field is clear and a Word value when the bit is set.

TRGOFF = D[W|D] Target offset.

This field is an offset into the specified target Object. It is not present when the Source Type specifies a 16-bit Selector fixup. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

```

+-----+-----+-----+-----+
00h | SRC |FLAGS|SRCOFF/CNT*|
+-----+-----+-----+-----+
03h/04h | MOD ORD# *| PROCEDURE NAME OFFSET*| ADDITIVE * @ |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.

@ These fields are optional.

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

PROCEDURE NAME OFFSET = D[W|D] Offset into the Import Procedure Name Table.

This field is an offset into the Import Procedure Name Table. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

```

+-----+-----+-----+
00h | SRC |FLAGS|SRCOFF/CNT*|
+-----+-----+-----+
03h/04h | MOD ORD# *|IMPORT ORD*|      ADDITIVE * @      |
+-----+-----+-----+
        | SRCOFF1 @ |      . . .      | SRCOFFn @ |
+-----+-----+-----+

```

* These fields are variable size.

@ These fields are optional.

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

IMPORT ORD = D[B|W|D] Imported ordinal number.

This is the imported procedure's ordinal number. It is a Byte value when the '8-bit Ordinal' bit in the target flags field is set. Otherwise it is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the

target flags field is clear and a Dword value when the bit is set.

```

+-----+-----+-----+-----+
00h | SRC | FLAGS | SRCOFF/CNT* |
+-----+-----+-----+-----+
03h/04h | ORD # * | ADDITIVE * @ |
+-----+-----+-----+-----+
| SRCOFF1 @ | . . . | SRCOFFn @ |
+-----+-----+-----+-----+

```

* These fields are variable size.
@ These fields are optional.

ENTRY # = D[B|W] Ordinal index into the Entry Table.

This field is an index into the current module's Entry Table to specify the target Object and offset. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

Import Module Name Table

Import Module Name Table

The import module name table defines the module name strings imported through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import module name table subtract the import module name table offset from the import procedure name table offset. These values are located in the linear EXE header. The end of the import module name table is not terminated by a special character, it is followed directly by the import procedure name table.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

```

+-----+-----+-----+-----+ +-----+
00h | LEN | ASCII STRING . . . |
+-----+-----+-----+-----+ +-----+

```

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 127 characters.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is

case sensitive and is not null terminated.

Import Procedure Name Table

Import Procedure Name Table

The import procedure name table defines the procedure name strings imported by this module through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import procedure name table add the fixup section size to the fixup page table offset, this computes the offset to the end of the fixup section, then subtract the import procedure name table offset. These values are located in the linear EXE header. The import procedure name table is followed by the data pages section. Since the data pages section is aligned on a 'page size' boundary, padded space may exist between the last import name string and the first page in the data pages section. If this padded space exists it will be zero filled.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

```
+-----+-----+-----+-----+ +-----+
00h | LEN |   ASCII STRING   . . . |
+-----+-----+-----+-----+ +-----+
```

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 127 characters.

The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Special pages

Preload Pages

The Preload Pages section is an optional section in the linear EXE module that coalesces a 'preload page set' into a contiguous section. The preload page set can be defined as the set of first used pages in the module. The preload page set can be specified by the application developer or can be derived by a tool that analyzes the programs memory usage

while it is running. By grouping the preload page set together, the preload pages can be read from the linear EXE module with one disk read.

The structure of the preload pages is no different than if they were demand loaded. They are non-iterated pages. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded.

All pages begin on a PAGE OFFSET SHIFT boundary from the base of the preload page section, as specified in the linear EXE header. The pages are ordered by logical page number within this section.

Note: OS/2 2.0 does not respect preload pages. Performance tests showed that better system performance was obtained by not respecting the preload request in the executable file.

Demand Load Pages

The Demand Loaded Pages section contains all the non-iterated pages for a linear EXE module that are not preloaded. When required, the whole page is loaded into memory from the module. The characteristics of each of these pages is specified in the Object Page Table. Every page begins on a PAGE OFFSET SHIFT boundary aligned offset from the demand loaded pages base specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded. The pages are ordered by logical page number within this section.

Iterated Data Pages

The Iterated Data Pages section contains all the pages for a linear EXE module that are iterated. When required, the set of iteration records are loaded into memory from the module and expanded to reconstitute the page. Every set of iteration records begins on a PAGE OFFSET SHIFT offset from the OBJECT ITER PAGES OFF specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. The pages are ordered by logical page number within this section.

This record structure is used to describe the iterated data for an object on a per-page basis.

```

+-----+-----+-----+-----+
00h |#ITERATIONS|DATA LENGTH|
+-----+-----+-----+-----+
04h |DATA BYTES | . . . | ... |
+-----+-----+-----+-----+
```

Figure 19. Object Iterated Data Record (Iteration Record)

#ITERATIONS = DW Number of iterations.
This specifies the number of times that the data is replicated.

DATA LENGTH = DW The size of the data pattern in bytes.
This specifies the number of bytes of data of which the pattern consists. The maximum size is one half of the PAGE SIZE (given in the module header). If a pattern exceeds this value then the data page will not be condensed into iterated data.

DATA = DB * DATA LENGTH The Data pattern to be replicated.
The next iteration record will immediately follow the last byte of the pattern. The offset of the next iteration record is easily calculated from the

offset of this record by adding the DATA LENGTH field and the sizes of the #ITERATIONS and DATA LENGTH fields.

Debug Information

Debug Information

The debug information is defined by the debugger and is not controlled by the linear EXE format or linker. The only data defined by the linear EXE format relative to the debug information is it's offset in the EXE file and length in bytes as defined in the linear EXE header.

To support multiple debuggers the first word of the debug information is a type field which determines the format of the debug information.

```
00h  01h  02h  03h  04h
+---+---+---+---+---+---+---+---+
| 'N' | 'B' | 'O' | n |  DEBUGGER DATA . . .
+---+---+---+---+---+---+---+---+
```

TYPE = DB DUP 4 Format type.

This defines the type of debugger data that exists in the remainder of the debug information. The signature consists of a string of four (4) ASCII characters: "NB0" followed by the ASCII representation for 'n'. The values for 'n' are defined as follows.

These format types are defined.

```
00h = 32-bit CodeView debugger format.
01h = AIX debugger format.
02h = 16-bit CodeView debugger format.
04h = 32-bit OS/2 PM debugger (IBM) format.
```

DEBUGGER DATA = Debugger specific data.

The format of the debugger data is defined by the debugger that is being used.

The values defined for the type field are not enforced by the system. It is the responsibility of the linker or debugging tools to follow the convention for the type field that is defined here.

Program Loader Architecture

This page is intentionally left blank.

Problem Description/Objectives

The major thrust toward the 386 is to allow programs to gain performance and reduce complexity. The 286 limits a single segment to 64K bytes. This requires all code and data that is larger than 64K to be separated into multiple segments. The manipulation of multiple segments results in a performance loss along with additional code complexity. The 386 helps reduce these problems by providing segments (now called objects) larger than 64K bytes. Code and data segments on the 386 may theoretically be up to 4G bytes in size.

To support this use of a linear executable image, a page based memory model is going to be used. The memory management will be based on a 32-bit linear addressing format called 0:32. The segment based system of OS/2 v1.0/1.1/1.2 used a 16-bit selector and 16-bit offset memory addressing format, referred to as 16:16.

The Program Loader component is responsible for loading application and library modules from a file into memory. The Program Loader is invoked by the tasking component during the execution of a new program to load the program module and its referenced library modules. It is also invoked by the runtime Program Loader API functions for runtime loading and linkage. It must also perform demand loading of pages for the Page Manager and cleanup of referenced modules during program termination.

The work required for the loading of a 0:32 module is for the most part defined by the format of the executable file image. The executable file image format is defined by the 32-bit Linear Executable Module document.

The definition of the Program Loader's function is rather simple; it must provide the loading of both 16-bit and 32-bit EXE file formats. The loading of the 16-bit segmented EXE files must be done in a manner compatible with the existing 286 based OS/2 products.

The loading of 32-bit EXE files must provide the following:

- Allocation of 32-bit objects.
- Recognition of the new linear executable module format.
- Use of the performance improvements made to the EXE file image:
 - Minimization of the number of disk reads performed when loading.
 - Coalescing of the fixup information into one table that is read in as part of the preload data.
 - Maintenance of fixup information on a per-page basis.
 - Loadtime and runtime linkage to both 16-bit and 32-bit entries in a dynamic link module.
 - Preloading of pages.
 - Demand loading of pages for the page manager.
- Support for stack-based program startup, DLL initialization, and DLL termination.
- Support for loading of Virtual Device Drivers.
- Support for loading of File System Drivers.

Solutions/Justification

The requirements for the Program Loader can be satisfied in at least two major ways. The first solution is to provide two separate loaders, one for the 16-bit EXE format and one for the 32-bit EXE format. This allows the 32-bit EXE format to be independent of the 16-bit EXE format.

This however would require the definition and management of two different data structures, one for each type of EXE format. Having separate routines for the manipulation of each format would greatly complicate and increase the size of the system loader code.

The second solution is to use a common loader for both formats. This is done by converting the 16-bit EXE to the 32-bit internal data structures. To simplify the use of a common internal data structure the 32-bit EXE format has been defined so that the 16-bit EXE header format may be easily converted into the 32-bit EXE header format. When a 16-bit EXE module is initially loaded its header is converted into the 32-bit EXE header format before it is linked into the other module data structures managed by the Program Loader. This allows some common code to be used for the management of program and library modules by the system. Using common code for the two module header formats simplifies the Program Loader, speeds it up, and reduces its code size.

The information not contained in the module header remains in separate formats to reduce the size of the internal kernel data structures. These data structures are still handled by some common routines that detect the differences between the 16-bit and 32-bit structure based on

an indicator in the common header data structure.

Architectural Review

To be performed.

Program Loader Design

This page is intentionally left blank.

Design Overview

The changes to the Program Loader to support the loading of the 32-bit Linear EXE format can be broken down into the following areas: the loading of modules, the allocation of objects, the preloading of pages, the demand loading of pages, the processing of fixups for a page, and the freeing of modules.

The loading of the 16-bit EXE must also be supported. That too can be broken down into the following areas: the loading of modules, the allocation of segments, the preloading of segments, the demand loading of segments, the processing of fixups for a segment, and the freeing of modules.

These two sets of functions are unique to the type of module being loaded. Some of the same methods can be used in doing both of them. So following is a description of each area, group together are the areas that can share common methods.

1. The loading of modules.

When a request to load a module is made, the modules that are already loaded are searched to check if the desired module is among them. If not, the EXE file is opened and the file header is checked to insure that it is valid. The current 16-bit EXE file must contain the value "NE" in the signature bytes. To insure that the 16-bit OS/2 system will not erroneously load a 32-bit EXE module, the 32-bit EXE file must contain the value "LE" in the signature bytes. The module header is initially read into a buffer the size of the new 32-bit header. The signatures are validated to see what type of EXE we are loading.

Each module that is loaded is managed by the system through a module table entry (mte) data structure. The mte contains:

- An executable information section.

This section contains information about the format of the executable. For example, the EXE signature, the byte and word ordering, and the executable format level.
- A module information section.

This section contains information specific to this module, like its version, flags and initial register values.
- A loader information section.

This section contains information that allows the Program Loader to determine the size of the data that will be read into memory.
- Table information section.

This section contains all the offsets to the required tables in the executable. The following is a list of them:
 - a. object tables or segment table
 - b. resource table

- c. resident name table
- d. entry table
- e. loader directives (Only valid for 32-bit module)
- f. per-page checksum (Only valid for 32-bit module)
- g. fixup tables (Only valid for 32-bit module)
- h. import module name table or module reference table,
- i. import procedure name table or imported name table,
- j. data pages (Only valid for 32-bit module)
- k. non-resident name table
- l. debug information

After the module header has been read in to an allocated memory object, the signature is checked to see if we are loading a 16-bit EXE file. If it is a 16-bit EXE, then the 16-bit EXE header is expanded to a 32-bit EXE header.

To limit size of memory used by preload pages, a check should be done before the mte is allocated. The value of preload pages found in the 32-bit EXE header will be compared to a maximum allowable value. If it is larger, then the maximum allowable number of preload pages will be read in.

Next the size of the mte is determined; this is done with a simple computation with fields from the 32-bit EXE header. Since the 16-bit EXE header was expanded into the 32-bit EXE header, the type of module that is being loaded does not affect the computation of the size of the mte.

The MTE is divided into two parts - the resident and the swappable. The resident part of the MTE consists of the module header, the pointers to MTEs it references and the full pathname of the file it is loaded from. The swappable part consists of the loader section: (object table, object page map table, object iterated data map table, resource table, resident name table, entry table) and the fixup section: (fixup page table, fixup record table, import module name table, import procedure name table).

The mte module header is initialized from the object containing the EXE header. The EXE header is simply copied into the mte. A few new fields are added to the end of mte header starting from the end of the EXE header in the reserved pad structure. They are a link to the next mte, a pointer to module pointers table, a count of the modules using this mte, the heap size for converted 16-bit modules, the stack size for converted 16-bit modules, the align shift for converted 16-bit modules, the mte handle, the file system number for the open file, a place to store a dynamic trace value, the number of preload pages read, the number of preload pages used, a pointer to the preload pages, the start of the object containing the preload pages and the size of the object containing the preload pages.

The remainder of the in-memory EXE file image, the section from the beginning of the segment table or object table through the end of the entry table or fixup tables plus the preload pages, is then read into another allocated system memory discardable object. Since the start of the preload pages must be a page boundary, padding may be added to the beginning of the object to force the preload section to start on a page boundary. The size of this read is computed from values out of the exe header plus the padding.

Now the in-memory loader and fixup sections, if needed, will be copied from this memory object into the mte in the swappable and non-swappable system heaps. After this copy is complete, the part of the object containing the Program Loader resident tables will now have the the physical memory released. After the preload pages are moved into the correct user space, the VMM is called to release the object. The segment table entry (ste) format for 16-bit modules has the addition of a segment handle field, and a linear address field. Each ste is moved into place and the two additional fields are added and initialized to zero.

The offsets in the mte header to the remaining tables are then updated to reflect their final offset. The update for a 16-bit module also requires the addition of a constant for the difference between the 32-bit EXE header and the 16-bit EXE header. Since the pointers in the EXE header are relative to the beginning of the header and space has been added between the EXE header and its tables when moved to the MTE. The pointers now must be updated by a constant value.

The module pointer table is initialized to zero. After all of the segments or objects in the module have been allocated, each module defined in the module reference table or import module name table is located or loaded, the pointer to the mte is stored in the corresponding position in the module pointer table. Finally, the module pathname is copied from a temporary buffer into the MTE.

2. The allocation of segments or objects.

After the module has been loaded, each segment or object is either allocated or attached to. For shared segments and objects that have previously been allocated, an attachment is made to the existing segment or object. For segments, this attachment simply allocates a selector in the desired process' LDT and maps it to the existing segment. When attaching to a shared object, a VMM call is made to attach to that object. When the Program Loader allocates a segment or object, the linear address is reserved, and the required overcommitment calculations are made.

The Program Loader will pass the VMM the needed VM flags, the size of memory to allocate, the mte handle, a linear address and a Module Count. This Module Count is a value used by the Program Loader to identify pages for 16-bit and 32-bit modules. The Page Manager will increment and store this value for each page when it is allocating memory for an object. Also the Page Manager will pass this value to the Program Loader when a page fault occurs. See demand loading of pages for a description of the use of the magic cookie.

A 16-bit module does not contain the required information to allow demand loading of pages within a segment. The fixup information for a 16-bit module unlike a 32-bit module resides in each segment after the data for that segment. So a complete read of the data for each segment must be completed to get to the fixup information for all the pages of that segment. When processing

the fixup information, chained fixups contain only the start target address of the chain. Contained in each target is the address of the next target. The Program Loader must follow the chain to apply each fixup in the segment by saving the next target and overwriting it with the source address. If the Program Loader was to apply the fixup information for a page in the middle of a fixup chain the information to follow that chain would be lost. This is unlike the fixup information for a 32-bit that can be gotten for any page with a simple lookup into the fixup page table. Were the information for a chained fixup kept with all the target address.

For 32-bit modules, object's which are writable and non-sparse are allocated as swappable. Read-only objects are allocated as discardable. Sparse objects are allocated as invalid. After all modules that are referenced are loaded, the objects are scanned to see if they contain any preload, invalid, or zero filled pages. If any of these type of pages are present, then the Program Loader must process each OPME. As the Program Loader checks the per-page attributes for each OPME, a call to the VMM is issued for pages which are not discardable or swappable. This memory call will change the memory from discardable or swappable to zero filled or invalid depending on the objects flag field. Also, the per-page attribute is checked for a legal physical page in the executable. If the page is present, a computation is done to see if it is a preload page. When a preload page is found, a routine will be called to load it.

3. The preloading of segments.

For a 16-bit module, a segment that is marked to be preloaded is allocated specifying preload to the VMM. When the module loading and segment allocation is complete, the Program Loader calls the routine LDTPreload in the Selector Manager to preload all of the segments that have been allocated with that attribute. The Selector Manager performs this preload by walking the LDT for this process, and each selectors marked preload will cause the Selector Manager to call the Program Loader interface LDRGetSegment to load that segment. The Selector Manager will pass the mte, the load address, and a Module Count. The Program Loader locates the ste for the segment by using the Module Count; the Module Count is shifted right 4 places yielding the index to the STE. With this, the Program Loader can load the complete segment.

4. The preloading of pages.

Unlike the method for 16-bit modules, the loading of a 32-bit preload page is handled by the Program Loader. During the processing of each object the page map entries are scanned, if a preload page is present, the Program Loader must preload it. To preload it the Program Loader must first try to lock the page down to see if it is still present. The reason this must be done is that preload pages were read into discardable memory during the initial loading of the module. Such pages could be discarded by the pager during the course of the load. If the page is still present in memory the fixups are applied, and the page is remapped to address which was passed in; this is done with a call to the Page Manager. If the page has been discarded, the Program Loader will not load it, and that page will be loaded later if it is referenced. After the preload of the preload pages is complete the Program Loader will release the linear address space used in the MTE for them.

5. The demand loading of pages.

The Program Loader is called by the Page Manager when a not-present page fault occurs. The Page Manager allocates a physical page prior to calling the Program Loader. The Program Loader is passed the mte handle, a Module Count, and the linear address of where to load the page. The Module Count is used differently for the two types of modules that can be loaded. For 32-bit modules, the Module Count is a logical page number, we can use this logical page number to index into the OPME. From the OPME, we may either get a physical page number or an OIDMO. With the physical page number, the data from the executable file and the possible fixup records are accessible to load this page. Using the OIDMO, the page can be loaded using iterated data.

If the module that the page fault occurred on was a 16-bit module, the high 12 bits of the magic cookie are the segment table index and the low 4 bits are the page number relative to the start of the segment. The magic cookie will be used to get the STE. From the STE, the file offset to the segment data is accessible. A buffer will be allocated, the segment data will be read, and the fixups applied to load this segment. Using the Module Count again, we get the page that the fault occurred on, we remap just this page into the faulting address space. The rest of the pages of the segment can be placed in the PF cache till needed or discarded.

6. The Processing of fixups for a page.

When the loading of the data for a 32-bit module page is completed, the Program Loader must check to see if any fixup records exist for this page. The physical page number obtained from the OPME will be used to index into the Fixup Page Table. Each FPT entry is an offset into the Fixup Relocation Table. To get the size of the fixup records for this page use the following algorithm:

size = FPT entry[page # - 1] - FPT entry[page #]

When the size returned by the algorithm is greater than 0, fixup records exist for this page. The Program Loader will process all fixup records till it reaches the next page's fixup records.

For 16-bit modules the fixup records are found after the segment data and can not be used on a per-page basis. See section on allocation of segments or objects for explanation of why a 16-bit module can not be loaded on a pre-page basis.

7. DLL Termination.

OS/2 v 2.0 supports DLL Termination routines for 32-bit DLLs. A 32-bit DLL may specify in its header that its entry point should be called when it is detached from a process as well as when it is attached to a process. The DLL must have a flat entry address. This feature is not supported at all for 16-bit DLLs. When a 32-bit DLL is entered via a 32-bit entry point, it is passed on the stack the handle of the DLL and a flag indicating whether the DLL is being initialized or terminated. A termination call is made only when

a process is about to detach from the module because of a FreeModule call. In the event of detachment because of a process' death, the DLL's entry is not called. While this behavior is not orthogonal with the behavior of Instance Library Initialization, it has been chosen because it is cleaner for NT OS to support. The argument is as follows:

- a. NT's model of thread termination does not allow for a terminating thread to return to user space because, in passing control back to user space, the system loses its ability to guarantee that the thread will actually terminate. Thread termination is meant to be a reliable operation.
- b. NT does not want DLL Termination to be implemented in the OS/2 layer because it would mean either exporting NT's module graph to the OS/2 layer or exporting some interface to the OS/2 layer that would allow it to make queries about the state of the graph, e.g., what is the set of affected modules if module X is freed?
- c. Since returning to user space from a FreeModule call does not violate thread termination model, NT would be able to support DLL Termination at FreeModule time without much trouble.
- d. NT believes that the long-term model of subsystem implementation is client-server in which there is no exact analog for DLL Termination, but in which the abstract problem of detaching a client from a subsystem can be solved cleanly by message passing. Thus, if the semantics of DLL Termination are a little clumsy, it does not matter, since people will be moving away from implementing subsystems as a set of DLLs. Given that a module can have both an ExitList routine and a DLL Termination routine, there is no inherent loss in power by not calling termination routines at process exit.

As with module loading and initialization, an attempt to free a module from ring 2 will fail if there are any termination calls to perform. In order to prevent races when termination calls are pending, a DosLoadModule call made by a thread of the process that is not executing a termination call will be blocked, while a DosLoadModule made by a thread that is will return an error.

Note that if a module receiving a termination call has an ExitList routine that it fails to deregister, the system is exposed to the following kind of bug:

- a. Process P does DosLoadModule on DLL D0. D0 asks to be called when it is freed. In its initialization routine, D0 registers an ExitList routine.
- b. Process P does DosFreeModule on DLL D0. There are no other processes attached to D0. In its termination routine, D0 neglects to deregister its ExitList routine. D0 is freed and its address space is available for reuse, but its ExitList routine is still registered with P.
- c. Process P does DosLoadModule on DLL D1. D1 is loaded where D0 used to be.
- d. Some number of other processes also attach to D1.
- e. Process P exits. The ExitList routine for D0 happens to lie in D1's code. Process P jumps randomly into D1 and manages to hang there, deadlocking itself and all the other processes attached to D1.

This sort of bug could be prevented by forcibly deregistering a DLL's ExitList routines, if any, after its DLL Termination routine has finished executing. However, NT does not wish to provide a mechanism for its OS/2 layer to perform this task, so OS/2 v 2.0 does not do it either in order to avoid any future compatibility problems.

8. The freeing of modules.

When a module is freed, the remaining shared segments or objects in the module must be freed by the Program Loader. The Program Loader will scan the segment table or object table and free the shared segments or objects that have non-zero handles in their table entry. This operation was previously not required by the Program Loader and is being added simply as a performance improvement during the termination of a task.

9. Virtual device driver loading.

For the loading of VDD modules, which only can be a 32-bit module, an interface to allocate memory from the VDM Manager is used. The Program Loader will use this memory in the same way as memory allocated from the VMM.

Data Structures Description

Exported Interfaces

Imported Interfaces

To be added.

Design Constraints

To be added.

Design Review

To be performed.

Program Loader Implementation

This page is intentionally left blank.

Internal Interfaces

The following internal procedure interfaces support the loading of module table entries.

internal procedure headers

The following internal procedure interfaces support the loading of segment table entries.

internal procedure headers

The following internal procedure interfaces support the API functions for runtime loading of library module.

internal procedure headers

The following internal procedure interfaces support the cleanup and freeing of modules.

internal procedure headers

The following internal procedure interfaces provide miscellaneous support functions with in the loader.

internal procedure headers

Loader data structures

```

exe_s    STRUC
exe_signature    dw    ?           ; Must contain "MZ"
exe_cblp         dw    ?           ; Count of bytes in last page of file
exe_cp          dw    ?           ; Count of 512 byte pages
exe_crlc        dw    ?           ; Count of reloc entries
exe_cparhdr      dw    ?           ; Count of paras in header
exe_minalloc     dw    ?           ; Min # of paras of BSS
exe_maxalloc     dw    ?           ; Max # of paras of BSS
exe_ss          dw    ?           ; SS of image
exe_sp          dw    ?           ; SP of image
exe_csum        dw    ?           ; Ignored checksum of file
exe_ip          dw    ?           ; IP of image
exe_cs          dw    ?           ; CS of image
exe_lfarlc      dw    ?           ; File address of reloc table
exe_ovno        dw    ?           ; Overlay number( 0 for root )
               dw    4 DUP ?      ; Reserved words
exe_oemid       dw    ?           ; Reserved for OEM's
exe_oeminfo     dw    ?           ; Reserved for OEM's
               dw    10 DUP ?     ; Reserved for future use
exe_lfanew      dd    ?           ; File address of segmented EXE header
exe_s    ENDS

EXE_VALID_SIGNATURE    EQU    "ZM"
EXE_VALID_OLDSIGNATURE EQU    "MZ"

```

MS-DOS and PC-DOS EXE Header Format

The EXE file format always begins with the MS-DOS EXE Header Format. This header describes the executable image used by the MS-DOS loader. This header is validated by the `exe_signature` field containing the value "MZ". When the `exe_rle_table` field is set to 0040h the `exe_nhdr_off` field is assumed to contain the file offset of the segmented EXE header. The remaining information in this header is not used by the segmented EXE loader.

The 32-bit Linear EXE Header describes the contents of a 32-bit EXE program or library file. The header is validated by the `mte_magic` field containing the value "LE".

The 16-bit segment table entry describes the segments to be loaded for a 16-bit module.

The 32-bit object table entry describes the objects to be loaded for a 32-bit module. The contents of each 32-bit object table entry is copied directly into an MTE object table entry when a module is loaded.

The 32-bit object page map specifies either a physical page or a set of data iterations for a logical page in an object.

The 32-bit Resource Table is used to identify resource that are contained within this module. Address to these resources are identified by their type and name.

The fixup record table contains entries for all fixups in the executable. The entries for any one physical page are group together in the table.

The 32-bit entry table contains object and offset information for ordinals entry points. Not all entry points in the entry table will exported, some entry points will only be used within the module. An ordinal number is used to index into the entry table. The entry table entries are numbered starting from one.

The following exported procedure interfaces support the loading of module table entries.

exported procedure headers

The following exported procedure interfaces support the loading of segment table entries.

exported procedure headers

The following exported procedure interfaces support the API functions for runtime loading of library module.

exported procedure headers

The following exported procedure interfaces support the cleanup and freeing of modules.

exported procedure headers

The following exported procedure interfaces provide miscellaneous support functions with in the loader.

exported procedure headers

comment from code

Implementation

To be added.

Implementation Review

To be performed.

Program Loader Appendix (included for each review)

This page is intentionally left blank.

Glossary

To be added.

Size and Performance Considerations

To be added.

Implementation Estimates

	LOCs	Effort	ELOCs	MM
API expansion to 32-bits Support module name being passed to the loader as a 32-bit value and searching for and opening the module file.	100	1.0	100	0.4
Recognize 32-bit EXE headers Simply recognizing the 32-bit EXE signature during the creation of the MTE.	20	1.0	20	0.1
Expand 16-bit header to 32-bit format Conversion of the 16-bit EXE header and segment table into the 32-bit format. This is all contained in create_mte.	200	1.0	200	0.8
Change mte and ste references to 32-bits Change all of the existing mte and ste references to the 32-bit EXE format. There are currently about 250 references to the mte and <100 references to the ste.	350	0.5	105	0.4
Allocate 32-bit object types Recognize the additional object type bits in the 32-bit EXE header and convert them to the interface required by the memory manager.	200	1.0	200	0.8
Preload pages	300	1.0	300	1.2
Demand load pages The loading of a single page.	150	1.0	150	0.6
Distiguish relocation record format The recognition of the expanded 32-bit relocation records and the handling of the additional 32-bit fixup types.	100	1.0	100	0.4
Distiguish module reference table	20	1.0	20	0.1

Handling of the 32-bit module reference table and module pointer table.				
Virtual device driver loading (open issues)	500	1.0	500	2.0
The work here is not yet well defined.				
Change the use of stack as temp data area for reading of EXE header info (old : new) and change to read in all preload data, fixup table scatter map, preload pages etc.	50	1.0	50	0.25
Add fixup table, scatter map, and directives table to create_mte.	20	1.0	20	0.1
Change way we process relocation records	50	1.0	50	0.25
Remove reading records on to stack.				
Add use of scatter map when loading segments (objects)	100	1.0	100	0.4
Remove all MemMapMTEAlias, and MemUnMapMTEAlias (17 references each)	40	0.5	20	0.1
Change interface with other memory management calls. (26 references)	100	1.0	100	0.4

Memory Subsystem

This page is intentionally left blank.

Page component

This page is intentionally left blank.

Page component Architecture

This page is intentionally left blank.

Problem Description/Objectives

This component is meant to meet several objectives:

- support the VMM.
- support memory objects larger than 64k.
- support memory objects larger than the available physical memory.
- support flat address model to enhance portability.
- support per-process linear arenas.
- support MVDMS.
- support LIM EMS specification memory emulation (with aliasing) for SAVDMs.
- reduce application startup time and improve the performance of typical applications that take advantage of virtual memory.

There is pressure from many sources to take advantage of the 386 support for large memory objects and multiple DOS machines.

Note: Some of the material contained in this document is superseded by the contents of the file [pgvp.doc](#). It should be read in conjunction with this text and in the future it should be integrated into this document.

Large Segments and Overcommit Accounting

Merely expanding the allowed segment sizes exposes serious limitations in the existing overcommit algorithm. Since the 286 causes segment not present faults when a segment register is loaded with a selector, an application cannot run unless the segments referenced by all four segment registers are simultaneously memory resident. In addition, execution of an instruction causing a segment not present fault cannot complete unless the newly referenced segments are also simultaneously memory resident. This requires that up to six 64k segments be made memory resident in order to execute an instruction that references two new segments (as in a ring 3 to ring 2 call gate transition). Because making several segments simultaneously memory resident is not an atomic operation and because the system only ensures enough memory is available to handle one task's full segment complement at a time, segment not present fault handling is globally serialized.

The combination of adding two new segment registers to the 386 and removing the 64k segment size limitation often means that there is not enough physical memory to hold the entire contents of all the segments an application wishes to simultaneously reference.

MVDM Considerations

DOS applications running in virtual 8086 mode reference memory without benefit of protected mode's segmentation; linear addresses are formed by a direct translation of application segment register contents. Since the linear addresses referenced in DOS machines conflict with other DOS machines, separate linear address spaces must be maintained. This implies that the Page Manager must be able to switch linear address contexts when the SAVDM context is switched. Since not every SAVDM consumes a full 640k of memory, it would be advantageous to postpone allocating SAVDM memory until it is actually referenced.

Solutions/Justification

It is suggested that the reader be familiar with the VMM design before proceeding.

The only way to support memory objects larger than the amount of physical memory is to take advantage of the 386's paging support. This also allows for a simple solution to the overcommit problem, because we only need to reserve a very small number of physical pages to allow at least one process at a time to perform useful work.

Conveniently enough, paging also allows for a separate linear address space for each protected mode and SAVDM context, and allows SAVDMs to allocate pages on demand. It also allows linear address aliasing, which is needed to support LIM EMS specification memory emulation and debugger aliases to a debuggee's memory.

Each protected mode linear address space is limited to 512 MBs, based on the 16 bit compatibility region's hardware limits of 8192 LDT selectors and 64k per segment. Future releases of the system will raise this limit by allocating memory objects that do not need to be accessed by 16 bit code outside of the 512 MB compatibility region.

Each SAVDM's linear address space is also limited to 512 MBs. Only the first 1 MB (plus the 64k wrap area above 1 MB) will be directly

accessible to a SAVDM application. The VDM Manager will place other memory objects and aliases in between the 1 MB and 4 MB boundaries. Also, LIM EMS specification memory objects will be allocated as needed above 4 MBs. The Page Manager will provide page directory aliases in the system linear arena, to allow the VDM Manager task time access to all SAVDM contexts up to the 4 MB boundary.

Linear Address Allocation

When a user memory object is allocated, linear address space is reserved and page table entries are allocated in the specified context; the page table entries are marked discarded or allocate-on-demand, based on allocation flags. Linear address space is always allocated when a memory object is created, but physical memory is only allocated as a result of page faults or lock requests.

Always allocating linear space allows two simplifications to be made to the VMM: all overcommit accounting can be moved into the Page Manager and code in the VMM that overloads not present descriptor address fields can be removed. Under previous OS/2 versions, the descriptor address fields contain the memory object handle when marked not present.

Page faults are entirely handled within the context of the faulting thread: in segment based versions of OS/2, context switches were required to communicate with a Swap manager task.

Aging Pages

A system thread will periodically examine all the pages in the system and mark not present any ptes that are present but not have not been accessed since the last scan. The priority of the aging thread will be dynamically variable to account for different memory request loads.

We plan to instrument the paging code to test the effect of:

- No SwapOuts in advance of the need for more physical memory. Force the faulting thread to call SwapOut when allocating a dirty page for its own use.
 - No in-fault-context swapouts. This requires the faulting thread to wait for page-aging to provide free or reclaimable pages.
-

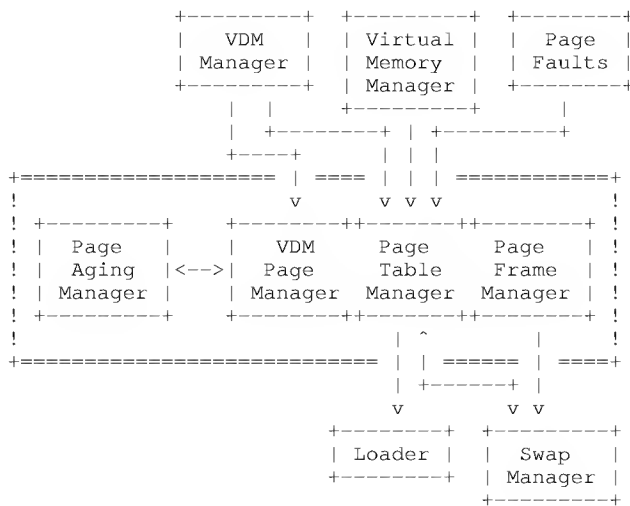
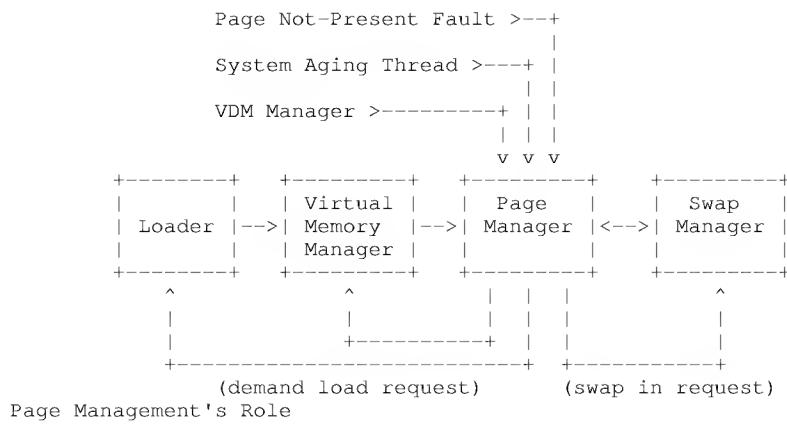
Architectural Review

To be performed.

Page component Design

This page is intentionally left blank.

Design Overview



Page Management Sub-components

The Page Manager is divided into four major components, the Page Table Manager, the Page Loading Manager, the Page Aging Manager, and the VDM Manager.

The Page Table Manager is responsible for:

- Page Manager initialization.
- Page Table Entry (pte) addressing.
- Page table allocation.
- Attaching to page table entries.
- Moving page table entries.
- Freeing page table entries.
- Locking page table entries and constructing lock handles.
- Flushing the TLB when appropriate.
- Linear to physical mapping.
- Handling page not present faults.
- Paging page tables.
- Page level serialization.
- Program Loader memory object allocation and preload support.
- Handling memory overcommit, reserving physical pages or swap space.
- Adding page tables into memory overcommit requirements.
- Page table context switching.

The Page Loading Manager is responsible for:

- Page Frame (pf) array allocation and initialization.
- Managing the Idle list and discardable page cache.
- Allocating physical pages in support of page not present faults.
- Freeing physical pages.
- Forcing physical contiguity and alignment in support of fixed allocation or lock requests.

The Page Aging Manager is responsible for:

- Aging pages.

The VDM Manager is responsible for:

- SAVDM page directory aliases.
- Linear motion for LIM EMS specification memory objects.
- Shadowing per-page dirty bits.
- SAVDM page faults.
- SAVDM cross-object locking and serialization.

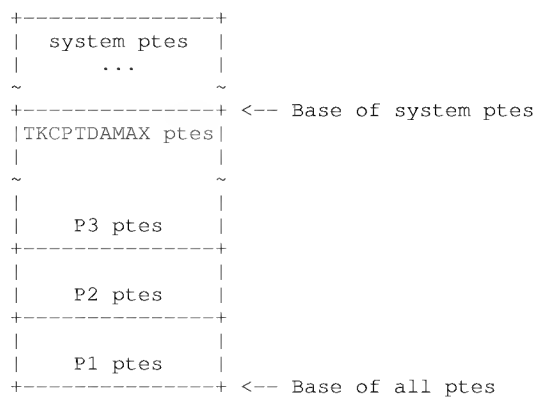
Page Table Management

This page is intentionally left blank.

Page Table Entry (pte) addressing

To address all page tables in a linear address space, enough linear memory will be reserved for page tables for TKCPTDAMAX maximally sized processes. The per process limit is 512 MBs, which amounts to 128 page tables per process (512m/4m per page table), or 512 KBs of linear address space in the system arena. Each task's PTDA will contain variables to point to the page tables for that task, and to indicate how many page tables are actually in use at the high and low ends of the 512 MB address space.

During initialization, enough linear memory is reserved for system page tables to map the maximum possible size of the system arena. Page tables and the page directory are allocated during initialization to map all memory that appears in the ArenaInfo array provided by SYSINIT. Further system page tables are allocated as needed when dynamic system memory allocations overrun the initial set of system page tables.



Method Chosen for Addressing Page Tables as Data

Page table allocation

During task creation, one of the blocks of page tables will be allocated for the new task. All exported routines accept ptدا pointer parameters

which will indirectly indicate which set of page tables to use. The ptdda pointer parameter is ignored for linear addresses in the system arena.

Page tables are allocated when linear memory is allocated. A small task will usually only need 2 page tables to run. Additional page tables are allocated as necessary if and when more linear address space is requested.

The following lists the types of memory objects for which ptes are allocated, along with a high level description of the data structure setup required (after ptes are allocated):

- Private (instance) read-write EXE/DLL data.

Ptes are marked not present, discarded, and swappable; the pte frame field is set to the Program Loader's Block No..

- Private API read-write data.

Ptes are marked not present, Allocate on-Demand, and swappable; the pte frame field is set to 0 (1 means zerofill).

- Shared EXE/DLL/API read-write data.

Ptes are marked not present, discarded, and swappable; the pte frame field is set to the Program Loader's Block No.

- Shared EXE/DLL code (or read-only data).

Ptes are marked not present, discarded, and discardable; the pte frame field is set to the Program Loader's Block No.

- System resident data.

Ptes are marked PRESENT and resident; a physical page is allocated, and the pte frame field is set to the physical page frame.

- System swappable data.

Ptes are marked not present, Allocate on-Demand, and swappable; the pte frame field is set to 0 (1 means zerofill).

Very similar results are obtained by changing the type of a page after the initial allocation.

No distinction need be made at the pte level between shared and private arena allocations. The only special case code that applies to system arena allocations is that the ptdda pointer parameter is ignored, and the system ptes are involved instead of the per task ptes.

Attaching to page table entries

A context that is creating a new reference to an existing memory object needs to "attach" to an existing set of page table entries. This could be in support of linear alias creation or attaching a new context to an existing memory object with shared contents or never-modified pages within a private object (Commit-on-write pages).

PGAttach will be used to allocate ptes in the new context and will copy an existing set of ptes, incrementing Page Manager data structure reference counts where appropriate. This is insufficient in the few cases noted below, because the two ptes meant to map the same data could eventually point to different physical pages. The old and new ptes must be made to point to a common structure to guarantee access to the same data. In the following cases, the attach code will allocate a swap frame for the page being aliased, and store state information in the swap frame:

- Shared or private Allocate on-Demand pages.

The Allocate on-Demand state is recorded in the SF flags word. Both the old and new ptes are edited to point to the newly allocated swap frame.

- Shared or private unmodified EXE/DLL swappable pages.

These pages can be discarded (and pass through the PF cache) up until the first time they are written to.

If the page is discarded, the discarded state is recorded in the SF flags word and the Block No. is stored in the SF's PF cross link field. Both the old and new ptes are edited to point to the newly allocated swap frame.

If the page is present, the newly allocated SF is cross linked to the PF. The new pte is edited to point to the physical page.

Moving page table entries

Linear motion is sometimes desired when an attempt to grow objects fails because adjacent linear space is not available. In a flat addressing model, linear motion is of dubious value because all pointers into the object would be invalidated and would have to be updated.

Linear motion between user memory objects will not be provided in the first release. Only a minimal interface will be provided in support of Program Loader preload pages. See the below section titled "Program Loader memory object allocation and preload support".

In a later release, a more general routine could be provided to move ptes from one linear address to another. Overcommit accounting would have to be performed when new page tables are required. If the overcommit accounting code determines the request cannot be satisfied, the motion request would have to be denied. The caller must, of course, be prepared for failure.

Freeing page table entries

When objects are freed, the VMM calls a provided routine to release the ptes. Each pte is examined to determine the current state of the linear page, and the Page Loading Manager is called to release any physical memory or Swap manager resources. The count of in-use ptes for the page table is decremented, and the page table itself is freed if the count goes to zero.

Locking page table entries and constructing lock handles

The greatest problem that exists in hiding paging from the VMM and old OS/2 device drivers is the use of DMA. DMA comes in underneath the paging mechanism and uses physical memory addresses. Previous OS/2 device drivers assume that memory objects are physically contiguous. But with paging enabled, memory objects are frequently not contiguous. Since we cannot easily change old device drivers at this point, we must ensure that any memory object to which a driver is doing DMA is contiguous in physical memory. Further, while DMA is taking place, we do not want any of the pages involved to be touched by the Page Manager, so they must be locked down.

The contiguity problem is solved by making any region of a memory object that may be involved in a DMA transfer temporarily physically contiguous. DMA may be done to a memory object that is allocated as fixed memory or to a locked region of a pageable memory object. Because not all lock requests require physically contiguous pages, the Page Manager provides a locking interface that optionally forces physical contiguity.

The locking interface only handles regions up to 64k in order to limit the expense of forcing contiguity and still fully support 16 bit segment locks. When fixed memory objects are allocated, physically contiguous pages are allocated.

A successful lock request returns a "lock handle", which may be subsequently passed to the unlock code where it is used to undo the lock. The lock handle specifies all the information required to perform an unlock.

From the Page Manager's point of view, a lock handle is a 96 bit structure which contains information about the locked region of memory: its base virtual page number, the number of pages locked, the object handle of the first (or only) object being locked, the ptda handle of the context to which the memory belongs, and a flag indicating whether the lock is long term or short term. This lock handle will be returned to callers of the new lock APIs, but the VMM will need to translate the 64 bit lock handle to a 32 bit value for users of the old style DevHlp calls.

Locks succeed when there will be 'enough' pageable memory after granting the lock request, and when the new lock doesn't conflict with an existing lock.

A new non-contiguous lock never conflicts with an existing lock. A new contiguous lock does not conflict with an existing lock when at least one of the following is true:

- It includes only a single page.
- It includes pages that are already contiguous.
- It includes pages that are all unlocked.
- The specific pages needed to complete contiguity are available (to completely lock a partially locked range of pages).

Locks can block inside the Page Manager waiting for a specific page to be made present or until enough pageable memory becomes available. Locks never block waiting for a long term lock to be released; an error is returned instead. In order to differentiate between short

and long term locks, separate lock counts are maintained for each physical page (in the PF structure). Locks that need to block until a specific short term lock is released, instead return a specific error indication and a blockid to allow the caller to release the memory object handle semaphore before blocking. For such unsuccessful lock requests, no effort is made to ensure FIFO order when reattempting locks.

Long term locks and locks that require contiguity are usually moved (the data are copied) to one end of physical memory to avoid physical memory fragmentation. Contiguous, short term locks of single pages are treated as though they do NOT require contiguity.

An attempt to free a page with outstanding short term locks will block until the locks are released. An attempt to free the pages of a memory object with outstanding long term locks will be failed with a special error code and no pages will be freed. During process termination, a process with long term locked memory will block until the locks are released and the memory may safely be freed.

Kernel managed resident memory objects need not be fixed or contiguous. This implies that individual PTDA's and Thread Control Blocks could straddle page boundaries, and that no kernel code can perform DMA-based I/O to the kernel stack. The special case of locking the local info seg (for write requests) could be handled by ensuring that it is entirely contained within the first page of the ptdata, and that the first page of each ptdata be effectively fixed.

Flushing the TLB when appropriate

Whenever a page directory entry or page table entry of a present page is modified, the CPU's TLB is flushed. If a set of related ptes are modified, the TLB is flushed once after changes are complete.

Linear to physical mapping

On a 286 or a 386 with paging disabled, linear addresses are equivalent to physical addresses. On a 386 with paging enabled, the address stored in a descriptor's base field is a linear address which (when memory is referenced) is translated to a physical address via the page directory and page table. Code dealing with descriptor base addresses must call procedures with linear address interfaces. Code dealing with DMA must call procedures with physical address interfaces.

The only routines that might use physical addresses are device drivers, and then only because they *might* attempt DMA. Many device drivers do not attempt DMA, but in order to keep the device driver interface consistent, all device drivers are passed physical addresses. The OS/2 kernel setup code for read and write requests calls VirtToPhys PGLinToPageList and passes the resultant physical address to the device driver in the request packet. Device drivers that use DMA can use the physical address directly, those that use virtual addresses for programmed I/O (as in "rep insw", etc.) call DevHlp_PhysToVirt, DevHlp_PageListToLin, or DevHlp_PageListToGDTSelector to translate the physical address into a virtual address.

For many ioctl requests, a virtual address is passed in the request packet. The device driver may, at task time in the context of the requesting task, use the virtual address directly (for non-DMA access), or call DevHlp.VirtToPhys or DevHlp_LinToPageList to obtain the physical address (for DMA or access at interrupt time or in a context other than the requesting task). For any of these routines to operate properly, the affected memory must be locked down.

The routines PGPageToPhys and PGLinToPageList may be used to query the physical address from a single page or range of pages, respectively. The reverse mapping can be performed using PGMapVirt (for physically contiguous ranges) or PGPageListToLin.

Handling page-not-present faults

The page fault handler determines the location of the faulted page's data. If the page is reclaimable, then the physical page is still available in memory, and has not yet been altered. The page table entry is modified to point to the physical page, and is marked present.

If the page is valid, but not reclaimable, the fault handler calls the Page Loading Manager to allocate a new physical page, then (based on the page state) restores the data by doing nothing, zeroing the page, or by calling the loader or swapper. After initializing the page contents, the page table entry is modified to point to the physical page, and is marked present.

The swap file space is not freed when swap ins occur, so that if the same page is to be swapped out again and has not been dirtied since the last time it was written, it needn't be rewritten to the swap file. It can simply be marked as though it were still out on disk and the physical page put directly on the idle list.

There are two sets of characteristics associated with each linear page, type and state. A page's type is associated with a linear page on a semi-permanent basis, generally for the life of the page; a page's state is much more transient.

Linear pages have one of the following types:

Invalid	never allocated.
Resident	permanently memory resident; not necessarily fixed or locked.
Discardable	discard when not referenced for some period of time.
Swappable	swap out when not referenced for some period of time.

When a page fault occurs, the Page Table Manager gets control, verifies the linear address by examining the corresponding page table entry, and takes the page semaphore. The fault handler takes appropriate action, based on the linear page state stored in the pte:

- Invalid - illegal to reference. Page faults in the context of a protected mode task are treated as GP faults (they are fatal to the process). Page faults in the context of a SAVDM task are passed off to the VDM Manager.
- Resident - permanently memory resident; not necessarily fixed or locked. No page faults occur.
- Present, pageable - can be referenced without causing faults. A data structure associated with the physical page frame indicates whether the page is swappable or discardable. No page faults occur.
- Discarded - the page is either discarded or cached in the PF cache. The Program Loader's Block No. is stored in the pte frame field, and is used to search the cache. If not cached, the Program Loader is called to restore the data.
- Swapped - the page is swapped, cached, discarded, or on the idle list (reclaimable). The Swap manager's SID is stored in the pte frame field, and is converted into a pointer to a structure associated with the swap frame, which is then examined to determine the page's disposition. If the swap frame's cross link field is NULL, the page is truly swapped out, and the Swap manager is called to restore the data. If the swap frame's cache bit is set, the cross link field holds the Program Loader's Block No., which is used to search the cache. If not cached, the Program Loader is called to restore the data.

Otherwise, the cross link field points to a physical frame structure, and the physical page is reclaimable.

- AOD (Allocate on demand) - allocate uninitialized page when referenced. If the pte frame field is non-zero, the physical page is zero initialized after allocation.

In order to support a 16:16 filesystem interface, the Page Manager reserves a number of linear address ranges to be used for disk I/O initiated by the Program Loader or Swap manager. Enough buffers are reserved to support several simultaneous page faults and since the Program Loader may be swappable, recursive fault must be accounted for as well. Most of the buffers are one page long, however a small number are 64kb in size to support the loading of whole segments from 16bit EXEs or DLLs. Permanent GDT selectors are used to map the reserved linear memory.

The page fault handler allocates physical pages and linear space from the reserved region, edits the appropriate ptes to make the physical memory addressable, then passes the linear address to the Program Loader or a 16:16 address to the Swap manager. The Program Loader needs to use the linear address to perform relocation fixups, and can call the Page Manager to translate the linear address into the corresponding GDT selector and 16 bit offset.

Paging page tables

The meaning of page directory entries' dirty and accessed bits is not defined by the hardware. Therefore, an active page table is not aged via the normal mechanism; a page table is marked dirty and placed on the idle list when every page mapped through the page table has been made not present.

To support this, a per-page-table count is used to keep track the number of pages mapped by a page table that are marked present. When this count becomes zero, the page table is moved onto the idle list for eventual swapout.

A second per-page-table count is used to keep track of the number of page table entries in use, so page tables can be freed when linear space is released.

A third per-page-table count is used to keep track of the number of resident, long-term-locked or system uvirt pages are on a page table. When a page table has a non-zero resident count, it is overcommitted for as a resident page.

Since page tables can be swapped out, some mechanism is needed to ensure page tables are available when the Page Table Manager needs to examine or modify their contents.

Internal Page Manager procedural interfaces are provided to pin down and release page tables for a given range of linear addresses. The pin routine will reclaim page tables or force them back into memory, as appropriate. When the page tables are later released, the above per-page-table counts are examined to see if the page table should be freed or placed on the idle list.

Page level serialization

Page semaphores may be obtained when pages are in any state; busy and wanted bits are defined in the pte, PF, and SF structures. The PF bits are used if a PF structure is associated with the page. Failing that, the SF bits are used if a SF structure is associated with the page. The pte bits are used when neither a PF or SF structure is available. If the busy bit is set, the requesting task sets the wanted bit and ProcBlocks, using the kernel linear address of the structure containing the semaphore bits as the block ID. If the wanted bit is set, and a change in the state of a page changes the location of the semaphore bits, a wakeup is performed on the old block ID. It is the responsibility of the woken thread to revalidate its pointers.

No memory object handle semaphore interaction is required when demand paging; taking page semaphores is enough to protect kernel data structures. Threads that attempt simultaneous free or realloc shrink attempts during page faults cause their own threads to fault in user space, but do not cause any kernel problems.

Allocation is always safe, because no other thread has access to an object during its creation. Lock, unlock, realloc and free attempts block until the handle semaphore is available; realloc and free attempts also block (without taking the semaphore) until all short-term locks are released. When long term locks are in effect realloc and free attempts return an error. No thread should ever block waiting for long term locks to be released. See the VMM documentation for more details. Paging operations do not obtain or check the handle semaphore.

Program Loader memory object allocation and preload support

The Program Loader calls PGAlloc (via the VMM) for the initial memory object allocation.

The Program Loader also calls PGSetType for each range of swappable or discardable pages that are to be zero initialized, so that the Page Manager can handle faulting in such pages without calling the Program Loader. Read-only pages can be reloaded from an EXE file so they remain permanently discardable, writeable pages cannot, so they are made swappable. Each conversion to discardable or swappable pages causes further overcommit accounting to be performed, which can fail if resource limitations dictate.

No physical memory other than page tables is allocated during these calls, because doing so could cause paging activity that is best postponed until the page is referenced.

The Program Loader loads preload pages from an EXE/DLL into a discardable system memory object via a single disk read. In support of this, the Page Manager exports an interface to move a page from a system memory object to the appropriate offset in a EXE/DLL defined memory object. The Program Loader calls this interface for each preloaded page. If the system object page is still in memory (if it hasn't been discarded), it is moved to the EXE/DLL memory object, and the system object page is invalidated.

Whenever a page is allocated as discarded, the Program Loader's Block No. is stored in the Page Manager's data structures, either in the pte or in a SF structure. When a fault is taken on a discarded page, the handle is used to obtain the mte handle, and the handle, mte handle and Block No. are passed to the Program Loader to demand load the page.

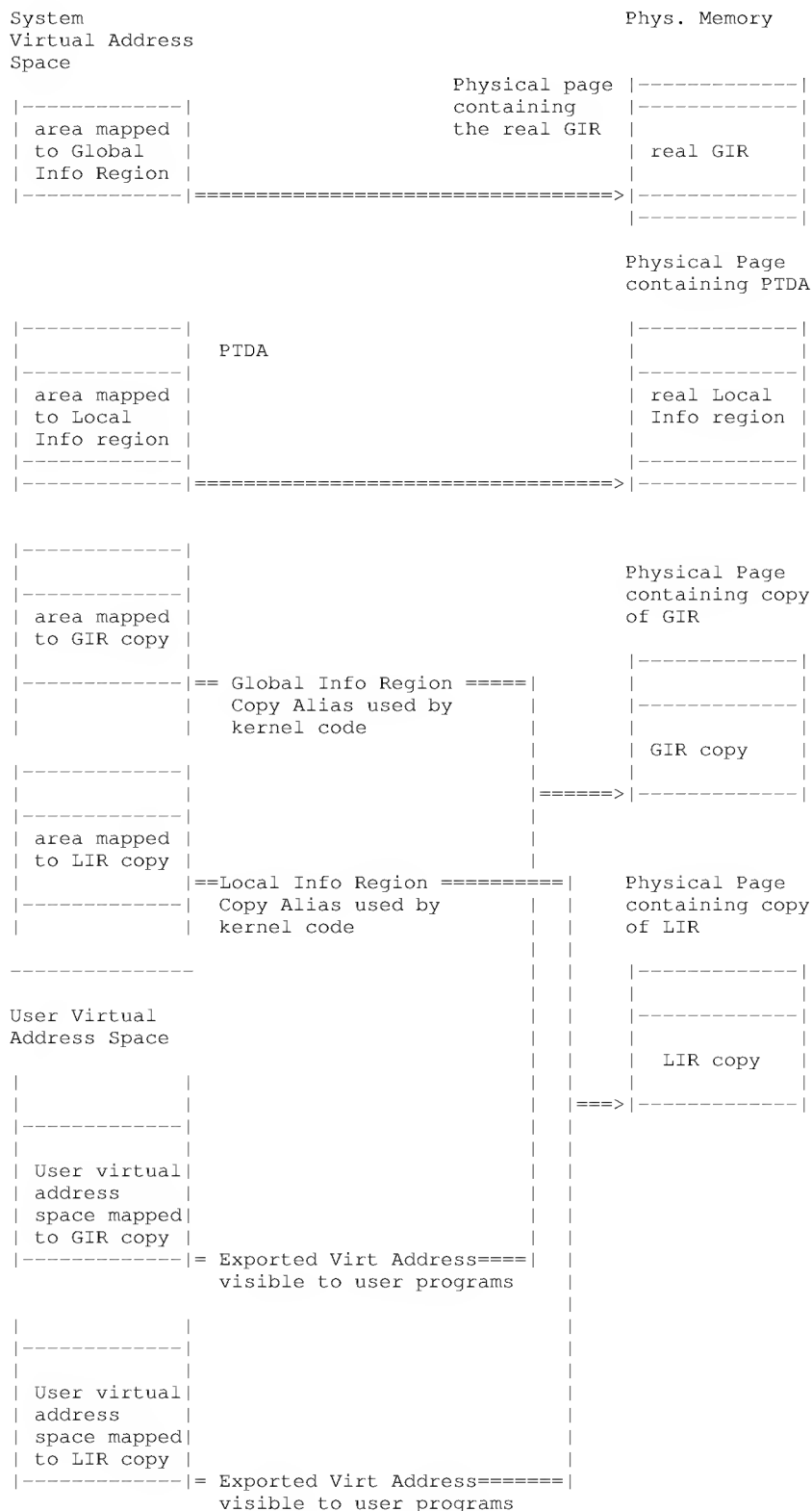
If the Program Loader detects the page is part of a 16 bit EXE/DLL segment that must be loaded all at once, the Program Loader returns a special error. This allows the Page Manager to interrogate the Program Loader for the size and location of the entire segment, in order to allocate enough physical memory to load the entire segment. The Page Manager then calls the Program Loader's segment demand load interface to satisfy the page fault.

Global and local information region support

Any thread of a task has read-only access to the Global Information Region and the Local Information Region for its task. As noted below, read-only access is implemented by giving the thread access to copies of the regions. The Global Information Region resides in a global data area within the system arena. The Local Information Region resides in a task's PTDA.

Two memory objects, one containing a copy of the Global Information Region and another containing a copy of the current Local Information Region, will be accessible to user programs. These objects will reside in the user virtual address space (for Cruiser, this is contained within the Compatibility Region; for Yawl this is not necessarily the case). The virtual address of the memory objects will be exported from DOSCALLS.DLL and will therefore be directly accessible to user programs.

The following diagram illustrates the proposed memory mapping.



Memory mapping for Global and Local Information Regions

Several design issues need to be considered

- Provision for Read Only Access

It is necessary to make a copy of the actual Global and Local Information for two reasons. First, marking the relevant physical pages as read-only and mapping them directly into user virtual address space will not suffice, since write accesses to that virtual address space from ring 2 are not inhibited by the read-only bit of page table entries. Second, the need to have the exported virtual address valid in all contexts conflicts with the design proposed for PTDA's. In that design, PTDA's will be implemented as regions of a memory object managed by BMP. Because of this, PTDA virtual addresses do not necessarily have the same virtual page offset and therefore the same would be true for the virtual addresses of the local information region.

- Access from Interrupt Time

The Global Information Region and its copy contain the current date and time and these must be accessible from interrupt-time code (namely the timer interrupt). To ensure such access, a virtual address alias from the system address space must map the physical page containing the copy at all times as a present page.

- Maintaining the Information Region Copies

Whenever user code is being executed, the Information Region copies must exactly match the real regions. The mechanism by which this tracking is performed must have minimal impact on performance. Two designs were considered.

- Design A - On an inter-task context switch, mark info pages not present

When a switch to a thread of a different task occurs, mark the page table entries for the pages containing the copies of the Global and Local Information Regions as not present. If any thread of the incoming task attempts to access one of the pages, a page fault will occur. Add special-case checks in the page fault handler to recognize attempts to access the "information pages". When such an attempt occurs, the handler marks the page as present and copies the appropriate Information Region.

Because the real Information Regions are updated on behalf of system calls, timer interrupts and other events, it will be necessary to either update the Information Regions and their copies at the same time or mark the pages not present before returning to user mode.

- Design B - On an inter-task context switch, copy the Local Information Region

When a switch to a thread of a different task occurs, the Local Information Region is copied from the incoming PTDA. In addition, whenever the current task's Local Information Region or the Global Information Region is updated, its copy will be updated as well.

- Justification for the Preferred Solution

Marking a page table entry not present at context switch time incurs significantly more overhead than copying the Local Information Region. Because Design B does not involve marking page table entries not present, it is the chosen design.

Handling memory overcommit, reserving physical pages or swap sp

```
fixedphyspages <= totalphyspages - pad
```

```
fixedphyspages + swappablepages <= SwapSpace + totalphyspages - pad
```

Overcommit Accounting Relations

All sizes are in pages:

- totalphyspages is the total amount of physical memory.
- fixedphyspages is the current amount of locked or fixed memory. This includes permanently resident non-fixed system memory, fixed memory, and memory locked for short or long term.
- pad is the minimum amount of physical memory that must be available for paging. This could be a constant number of pages or a constant percentage of totalphyspages.
- nswappablepages is the total amount of swappable virtual memory allocated. No accounting is performed for discardable memory; the pad reserved for paging is large enough to hold some minimum number of discardable pages.
- SwapSpace is the current swap file size.

When an allocation or lock request would cause fixedphyspages to grow enough to make the first relation invalid, the request is refused or postponed. When a request would cause fixedphyspages or nswappablepages to grow enough to make the second relation invalid, a (serialized) attempt is made via the Swap manager to grow SwapSpace (and the swap file). If the growth attempt fails, the request is refused.

The amount of virtual memory in a system has a practical limit of the sum of the actual physical memory and the available swap space (the storage of dirty pages). The difference between the amount of virtual and physical memory is the RAM overcommit capacity of the system.

Adding page tables into memory overcommit requirements

New page table pages are required when allocating an object that includes a linear address where no page table currently exists. These additional pages must be taken into consideration at the same time overcommit accounting is performed. The count of in-use ptes for each page table affected is examined. Those page tables with zero counts do not exist and must be allocated as a part of the allocation request.

All page tables are given swappable overcommit when they are allocated. If a page table maps any resident, long-term-locked, or system uvirt pages (i.e., its resident page count is non-zero) it is also given resident overcommit.

Page table context switching

Each PTDA holds an array of 128 page directory entries to be copied into the page directory when the task is made active. The PTDA also records the number of page tables in use at each end of the 512 MB compatibility region.

To effect a context switch, the descending PTDA's counts are used to zero all of the page directory entries used by the descending task. Then the ascending PTDA's counts are used to initialize the page directory entries needed by the new task. Since the typical task is expected to use very few page tables, this greatly reduces context switch time page directory editing.

Topics to be addressed

- Consider supporting user installable pagers. Do memory mapped files fall out of the installable pager support?
- Cache mtes and their corresponding sfts, allowing the PF cache to contain pages for unreferenced EXEs/DLLs across execs.
- Consider supporting memory mapped devices, context switching (e.g. Weitek 1167).

Page Frame Management

This page is intentionally left blank.

Page Frame (pf) array allocation and initialization

The page frame array is sized to contain one entry for each addressable physical page. It is carved out of unused linear and physical memory during initialization. Each entry that is associated with an in-use physical page as described by a SYSINIT data structure is initialized as a resident page. All other entries are initialized as free pages and are placed on the free list.

Managing the free and idle lists

Physical pages that are locked down or consumed as part of a resident memory object are not available for paging. Physical pages that are available for paging are subject to being aged and placed on the idle list.

The idle list contains in-memory pages sorted by LRU order that are not mapped through any valid (present) page table entries, i.e. their reference counts are zero and they are reclaimable. Reclaimable pages may be swappable or discardable, and clean or dirty. Clean swappable pages may have an up-to-date image on the swap media, dirty swappable pages do not. Discardable pages can always be reloaded from the EXE or DLL file from which they came, hence they are always clean. A page is placed on the MRU end of the idle list when it is removed from a process' working set by the page aging thread. A page is removed from the idle list when it explicitly freed; when it is made present in any context, via a page fault or lock; or when the page is swapped-out or discarded so the frame can be used for other purposes.

Physical frames not mapped by any page table entries that are non-reclaimable are considered free and kept on a separate free list. On machines with different speed classes of memory, the free list is kept sorted so that all of the "fast" pages are on one end of the list. Page allocations always take place from the "fast" end of the free list.

Reclaiming pages and allocating contiguous blocks of pages results in removing pages from arbitrary locations in the idle or free lists, whereas allocating non-contiguous pages always removes them from the fast end of the free list or LRU end of the idle list.

Allocating physical pages in support of page not present faults

To allocate a physical page, a free page is taken from the free list. If no free page is available, a reclaimable page from the idle list is used. If the page is swappable, its data are swapped out (if dirty or an image does not already exist on the swap media), the "pf valid" bit in the vp structure is cleared to indicate the page is no longer reclaimable and the swap file disk frame number is copied from the pf structure to the vp structure. If the page is discardable, the "pf valid" bit in the vp structure is cleared and the mte block number is copied from the pf structure to the vp structure.

A count of the number of pages on the idle and free lists is maintained, and when it drops below a low water mark, the page aging frequency is increased. Similarly, when the number of pages on the idle and free lists rises above a high water mark, the page aging frequency is decreased. When there are insufficient pages on the idle and free lists to satisfy a request, a flag is set indicating a thread is waiting for pages to allocate, and the thread blocks until one or more pages become available.

Forcing physical contiguity and alignment

The algorithm for making a region of memory contiguous proceeds as follows:

1. Scan the physical page array looking for a region of contiguous memory that is both available (all pages pageable and not busy) and sufficiently large. If such a region is not found, return `ERROR_NOT_ENOUGH_MEMORY`. Select the region with the largest number of free or idle pages to minimize the amount of work needed to do later.

At this point, we have an existing mapping from page table entries to physical pages (the source list) and a set of contiguous physical pages to which we wish to map those page table entries (the target list). We also own all semaphores in both the source and target regions. The next step initiates a loop through the list of page table entries and the list of target pages.

2. Allocate a free page for temporary use and take its semaphore.
3. Top of the loop. If we are at the end of the lists, exit the loop.
4. If the target page is free, then copy the data from the source page into it. Otherwise, copy the data from the target page into the free page, and copy the data from the source page into the target page. Note that the source page is now the free page. Release the semaphore belonging to what was the target page.
5. Update all ptes involved in the move (both source and target) so that they have effectively moved along with the page data.
6. If any pages were copied, flush the TLB and TCYield. Update loop variables and go back to the top of the loop.
7. Release the free page and its semaphore.

Note that we optimize the case of an object occupying a single page by simply locking that page and skipping this algorithm.

Page Aging Management

The OS/2 1.3 page aging code periodically scans all of the present pages of memory in the system and if they have not been accessed since the last scan they will be marked not present and placed on the MRU end of the idle page list. There they will be available for future reclamation or swapping, as needed.

The page aging code should have the following attributes:

- Global effects
- Real time aging
- Split work into packages that never cause significant latency effects
- Minimize system-wide performance degradation by:
 - Doing as much work as possible during idle time
 - Making each process responsible for its own aging
- Dynamically tunable aging rate to accommodate different memory request loads

Any aging system that meets these objects would be constructed from three logical units: an idle-time ager, a local run-time ager, and a global ager for blocked (or otherwise not run) processes to be used when there is not enough idle time. These three units should be used in the order listed above for maximum system performance -- aging should be done in idle time if possible; barring that, in a process' own time; and if worse comes to worst, the time must be taken from the system as a whole (i.e., from whatever process happens to be executing). Strategies for implementing these duties are discussed below.

Idle-time aging

Under most circumstances, aging will not be a time-critical activity and thus may be delayed for periods on the order of seconds. With this sort of leeway, it is possible to pick the ideal time to actually carry out aging and the obvious choice is when no other thread is in need of the cpu. During idle time, the aging code could age pages until another thread was ready to run (or until no more aging is needed). Two means of implementing the idle-time ager suggest themselves:

1. A separate system thread of very low priority (probably idle-time class) which would automatically be given cpu time by the scheduler only when no other thread was ready to run. This thread would be coded as an infinite loop that would look for unaged page tables and age them whenever it gained control. A separate aging thread would be the most flexible method to implement aging since it is isolated from the rest of the kernel. Scheduling the ager to run at the appropriate time would also be quite simple since the tasking code would do so automatically. However, using a new thread for aging would reduce the already limited number of threads available and would slow down the system with extra task switches. An alternative would be,
2. New code in the kernel idle loop in SchedNext. In SchedNext there is currently a very tight loop which calls `get_next_runner` and if

no runner is available it loops back for another try. After the `get_next_runner` call, code could be inserted to check if no thread is currently available and if not aging could commence. This would continue until `get_next_runner` returned an unblocked thread. The new kernel code would add complexity to what is now a very clean idle loop and would prevent future versions of the operating system from replacing the idle loop with a HLT instruction which would reduce bus traffic on a multi-processor system. On the other hand, this method would not consume as many valuable system resources (cpu time and scheduler slots) as a separate thread would.

Both of these methods might cause latency problems if a high priority thread became available to run in the middle of aging a large group of pages, but dividing the work into smallish packages followed by yields for the thread method or checks of `get_next_runner` by the kernel method should alleviate the condition.

In the first implementation of the page aging code, a separate thread will be used for idle time aging.

Local run-time aging

If enough idle-time is not available in the system, the most equitable way to carry out aging is for each process to pay for it from its own time slice. This could be done by aging the pages of a single process some time during its own time slice. The logical place to put the local aging code is in the kernel just before a thread is about to be run (such as in `Exit_Kmode`). A flag (a force flag or new flag that will act like one) would be checked to see if the process' pages have already been aged in the current aging sweep and if not, the process' pages would be aged and the flag cleared. However, having aging take place at the beginning of the time slice introduces a possibly serious problem -- if no threads of a process had run for a while, the aging might idle all or most of the process' memory just a few clocks before the memory will be needed. Since aged memory will not usually be physically swapped out at once it will be easily reclaimable, but the rampant page faults that might occur should be avoided none the less.

Placing the local aging code at the end of a thread's time slice would lessen that problem. However, there is no way to predict when a time slice is about to end, nor how much time must be reserved from that for aging. Any foreseeable method of aging at the end of a time slice would actually take the time from the NEXT thread to run, which of course would eliminate one of the main reasons for using local aging -- the fair distribution of the burden of aging.

Both of these methods of local aging might cause latency problems if a time critical thread is about to be run and the aging is budgeted from its time slice. Under such circumstances, turning off local aging might be the best solution. However, as aging must occur sometime, putting it off until later might cause even worse problems for the same time critical task later when memory resources get lower and more aggressive and time-consuming global aging must take place.

Abandoning local run-time aging entirely should be seriously considered as well. The idle and global aging code described above and below could supply the system with memory. The only drawback would be the elimination of one type of "fairness" (pay for your own aging) with another (each process pays for one equal chunk of aging). Unless mandated by performance considerations, local aging will not be used in the first release.

Global aging

The term "global aging" is being used loosely here to refer to the aging, process by process, of pages that have not been aged by other means. Its implementation is problematic because the procedure will cause general performance degradation in the system -- the only question to the designer is where and how to distribute this load.

The global ager will probably be constructed as part of the idle-time ager. If a separate thread is used for idle-time aging, that thread could be bumped up in priority until it gained control and would age the remaining unaged pages in a few passes between short yields. If an addition to the idle loop is used for idle-time aging, the same code aging could be used but it would be activated by a global aging flag rather than by no runner being available. Each method has the same advantages and disadvantages of its associated idle-time sibling -- a trade-off between design simplicity and resource usage. A thread-based global ager has the added disadvantage of being difficult to start-up, since there is no simple relationship between an individual thread's priority and its chance of running. However inexact, the scheduling mechanism already exists and would not have to be reinvented as it would for the other proposed method. For the sake logical simplicity, a separate thread has been chosen to implement the global ager.

The aging driver

Choosing the aging method to use at any one time (if any) is the work of what I will call the aging driver. The most conventional way to implement the aging driver would be to choose an interval of time as the "sweep rate" and every n clock ticks complete one aging sweep though all of the memory in the system that must be aged. For performance reasons, the first half or so of the sweep interval should be reserved for idle time aging and if that had not aged all of the processes in the system, the more aggressive global aging would take place for the rest of the interval.

The actual length of the aging interval could in theory be dynamically tuned by the system to match the memory request load. However, in practice this would be extremely difficult. The likely length of the average aging interval would probably be on the order of seconds while the page fault rate (which reflects memory consumption) has a significantly finer granularity. Translating one figure to the other accurately would be impossible. Since page faults tend to come in spurts followed by relative lulls, the chances are that the Page Frame Manager would constantly be under- or over-supplied with idle pages.

A better method would be to drive the aging from the number of pages currently on the idle list. If the idle list contained a great many idle pages no aging would take place. When the number of pages on the idle list fell below a low water mark, aging during idle time could be activated. When a second lower water mark is reached, the priority of the aging thread could be increased to an intermediate level. When a third very low water mark is passed, the aging thread could have its priority increased to time critical priority. Further water marks could also be included for critical memory shortfalls which would increase the number of pages aged per scheduling of the page aging thread. As the number of idled pages increased because of the increasing aggressiveness of the aging methods used, the water marks would be passed in reverse and the associated aging measures would be discontinued.

This method has the advantage of tying the aging rate directly to the memory needs of the system. Keeping track of the size of the idle list would be quite simple since all insertions and deletions from the list pass through two Page Frame Manager calls. Within these frame management calls checks could be made to see if a water mark is passed in either direction and if so it would call pgAgeRate which would increase or decrease the aggressiveness of the aging.

Process aging strategy

To narrow down the actual amount of memory the aging code looks at in each sweep, it would be possible to use the private and shared arenas to guide us to the memory that is not free. This approach, however, has its drawbacks: it adds an extra layer of indirection between the ager and the ptes it must examine; it tells us what memory is allocated but not what memory is present and actually needs to be aged; it would be impossible for the arena-walking code to block without restarting the walk at the beginning of the arena when reawakened. It would most probably be a performance win to skip this "optimization" entirely and use the page directory entries (pdes) to guide us to memory. The pde will be zero if no pages are allocated AND no pages are present within the page table, thus the page ager would only have to examine ageable memory. Using the pdes would also make it relatively simple to block and restart, since pdes are in a fixed size array rather than a linked list.

To keep track of which processes have already had their pages aged in the current aging sweep, a new per-process flag will be established in each ptda which is set when the process is in need of aging. These flags will be set for all processes once at the beginning of each sweep and will be cleared as they are aged, to prevent one process from being aged twice before another has been aged once.

Swappable system memory

The ager will treat system pages as if they belonged to a phantom process and will age them once per sweep. Since little of the system memory is likely to be swappable (at least for the near term) it might be a significant optimization to keep a separate list of swappable system memory and age it without examining all the system page tables. Examining all system page tables, however, is much simpler to implement and will probably offer acceptable performance today and will be the logical method of implementation in the future when more of the kernel is swappable.

VDM Page Management

This page is intentionally left blank.

SAVDM page directory aliases

When a SAVDM is created, the page manager routine PGInitVDMContext is called. This routine calls the VMM to allocate 4 megabytes of address space from the system arena on a 4mb linear boundary. The one page directory entry that maps that 4mb region is mapped to the same page table that describes the first 4mb of address space in the new SAVDM context. PGInitVDMContext returns the base address of the alias range, so the SAVDM manager can have task time access to all SAVDMs. The alias is freed when the page table context for the SAVDM is freed.

Linear motion for LIM EMS specification memory objects

The VDM Manager would like to grow LIM EMS specification memory objects when an application requests it. Since the growth will quite often collide with another memory object, moving the memory object in the linear address space to another location that does have enough free linear memory is desirable. The Page Manager will support this by allowing unlocked, unaliased ptes to be moved from one object to another. The destination pages must be unallocated pages inside a sparse object. After the move completes the source pages will be unallocated pages, effectively making the source object into a sparse object. All aliases and long term locks on the pages involved must be released prior to the call or it will fail.

Shadowing per-page dirty bits

The VDM Manager needs a routine to fetch and clear the dirty bits for linear address ranges. The Page Manager provides such a routine that operates on linear alias windows, physical aliases, and normal memory objects, ignoring memory object boundaries. Since the Page Manager cannot allow the dirty bit information to be arbitrarily cleared, the bit will be shadowed.

When the routine is called, the real dirty bit is propagated down to the shadow, and the shadow bit is returned and cleared. This preserves the integrity of the 'real' dirty bit, which will allow swapping to be done correctly. The aging code propagates the dirty bits in the same manner.

The interface returns a bit vector of up to 32 pages worth of dirty bits, and will return a 1 in a bit position to indicate that a page was written to in either of the following ways:

- through the linear address being interrogated, since the previous call.
- through another linear address, where aging was performed on the other linear address since the previous call.

This leaves a window where an undetected write could have happened through another linear address, because aging may not yet have taken place. The VDM Manager can live with this window.

When called on physical aliases, the pte's dirty bit will be returned and cleared; no shadowing is performed.

SAVDM page faults

Memory allocated on behalf of a SAVDM will usually be pageable. Page faults will normally be completely handled by the standard Page Manager page fault mechanism. When the Page Manager encounters an unresolvable page fault (invalid page or attempted write to read-only page) in the context of a SAVDM, the VDM Manager will be called with the faulting linear address. The VDM Manager can decide to handle the fault or terminate the application.

SAVDM cross-object locking and serialization

Because of the cross-object locking requirement, handle based serialization breaks down. Even though a SAVDM has only one thread, the VMM needs to serialize memory management operations because another context could request 'remote' memory allocation in the context of a SAVDM.

Because each SAVDM is made up of many different memory objects and aliases, and because the DOS application has no knowledge of the memory management environment, I/O requests will result in attempts to lock memory that may span multiple objects and/or multiple alias windows. Except for the following, these locks must be supported by the VMM and Page Manager.

The only problem remaining is that the Page Manager will fail a lock request on a range of pages that cross a memory object boundary when one of the objects is fixed and contiguity is requested. This will occur when unmapped DMA-based I/O is attempted to a memory region that crosses the 64k linear address boundary, and if running on a 386 stepping that requires the 64k fixed memory in each SAVDM context for an errata work around.

In this case, the I/O must be broken up by the VDM Manager. If the I/O does not require DMA, or if it uses scatter map DMA, the lock (and the I/O) will be performed transparently. If the I/O DOES require DMA to physically contiguous addresses, the I/O itself MUST be buffered or performed as two separate requests.

Data Structures Description

So that the page table manager can initialize and maintain the maximum possible number of page tables, one page table and 4 MBs of linear address space are reserved during initialization to access page tables as data. Additional page tables are allocated later as necessary when linear address space allocation takes place.

Each pte indicates whether the corresponding page is present, locked or has been recently accessed and modified. The state field (pt_state) indicates one of invalid, swappable, discardable, or permanently memory resident. If not present, the state field indicates whether the page is invalid, reclaimable, swapped, discarded, AOD, or AOD+zeroinit. If present or reclaimable, the frame field (pt_frame) contains the physical page frame: if swapped out, the SID, if discarded, the Block No..

The page frame array is sized to contain one entry for each addressible physical page. Just as for page tables, linear address space and page tables are allocated during initialization to map all of the page frame array through 32 bit near pointers. However, enough physical pages are obtained and mapped into the reserved linear address space to immediately initialize all of the page frame array.

Physical memory is not necessarily continuous--there may be holes. In order to provide the most efficient mapping of a page frame index to its corresponding pf structure pointer, the page frame array is allocated as a sparse object where the "holes" in the array correspond to the holes in physical memory. These holes in the array are donated to the kernel's resident heap so that the memory will not be wasted. Thus a pointer into the page frame array is obtained by multiplying a page frame index by the size of the pf structure and adding the base of the array.

Design Constraints

To be added.

Design Review

To be performed.

Page component Implementation

This page is intentionally left blank.

Implementation

To be added.

Implementation Review

To be performed.

Page component Appendix

This page is intentionally left blank.

Glossary

To be added.

Size and Performance Considerations

To be added.

Implementation Estimates

	LOCs	Effort	ELOCs	MM
Page manager initialization	400	0.5	200	0.8
Set up initial page tables, enable paging.				
Debugging support code	200	0.5	100	0.4
'C' printf support.				
Page table entry manipulation	550	0.5	275	1.1
Declaration and manipulation of PTEs.				
Page frame table manipulation	650	0.5	325	1.3
Declaration and manipulation of PFTs.				
MVDM support	300	1.0	300	1.2
Force contiguity in a range of v86 pages,				
page v86 pages, support special allocation				
requirements (errata workarounds).				
page-based overcommit	200	2.0	400	1.6
Add in page table requirements and pass				
request on to page-based swapper.				
dynamic page table allocation	300	1.0	300	1.2
Allocate linear space for PTEs.				
Set up array of GDT descriptors to map PTEs				
into 16:16 pointers for 16 bit 'C' code.				
new LinToPTE algorithm and initialization	100	1.0	100	0.4
Interpret GDT selector array for PTEs.				
Fix callers to use 16:16 pointers.				
dynamic (init-time) PFT allocation	200	1.0	200	0.8
Allocate linear space for PFTs.				
Set up array of GDT descriptors to map PFTs				
into 16:16 pointers for 16 bit 'C' code.				
redefinition of PTE and PFT field	400	1.0	400	1.6
Conversion of flag bits and structures to				
match our needs for swappable/discardable.				
expanded SegLock interface support,				
locklist management	500	1.0	500	2.0
page contiguity algorithm	700	1.0	700	2.8
Force pages to be physically contiguous for				
DMA.				
paging page tables	100	2.0	200	0.8
Ensure that page faults on page tables can				
be handled without problems.				
page-based semaphores	100	1.5	150	0.6
Take and release page semaphores when				
manipulating pages.				
page fault handler	100	1.0	100	0.4
Allocate page, call the appropriate handler				
(loader or swapper); tell loader if it's				
the first page fault within the object.				
page aging algorithm	700	1.0	700	2.8
Active, swap and idle list transition				
management.				
Total	5500		4950	19.8

Proposed Cruiser Page Manager Data Structure Revision

Proposed Cruiser Page Manager Data Structure Revision

1. Data Structures

1.1 The VP

Each committed page in the system is represented by a 10-byte Virtual Page structure (VP), which exists for the life of the page. The VP for any page contains:

A reference count of the number of PTEs that share the same page contents (whether marked present or not). (16 bits)

A handle to the Virtual Memory Manager Object Record (HOB) which can be used to enumerate all references of the page or query per-object information (such as the MTE handle). (16 bits)

A HOB-relative page number. (16 bits)

Flags (12 bits), which include:

VP_BUSY	- page semaphore taken
VP_WANTED	- page semaphore requested
VP_CACHE	- search page cache for pf (currently unused)
VP_PFIDLE	- cross linked to idle pf
VP_PF	- cross linked to pf
VP_DF	- has swap file disk frame
VP_DIRTY	- contents written to
VP_SHDIRTY	- shadow dirty bit (for VDMs)
VP_SOW	- change to swappable on write
VP_RES	- unused
VP_RESIDENT	- cannot be moved or swapped
VP_DISCARDABLE	- contents may be discarded and reloaded from file

A frame field (20 bits) which contains information about where an image of the page may be found.

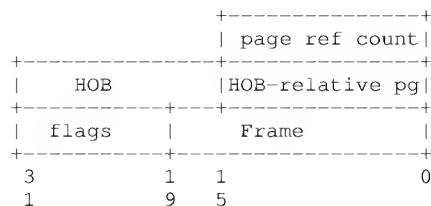
Possible values are:

If a valid image of the page exists in RAM, the field contains the physical page index (which is also an index into the PF array -- see below).

Else, if a valid image of the page exists on the disk, the field contains a page offset into the file in which the page exists -- either a swap file "DF" (see below) or loader "Block number".

Else, if the page is a swappable page that has never been referenced (allocate-on-demand), the field contains a flag as to whether we should zero-fill the page when making it present.

Figure 1. The VP



1.2 The PF

Each page of physical memory in the system is represented by a 12-byte Page Frame structure (PF). The PFs are stored in an array that is indexed by physical frame number. A PF may be in one of three states: in-use, idle, or free. A PF is in-use if any present PTEs in the system reference its physical page. An in-use PF (figure 2) contains the following fields:

A pointer to the associated VP. (32 bits)

A count of long-term locks outstanding on the page. (8 bits)

Flags (4 bits) which describe:

PF_FAST	- frame is in fast memory. Set during initialization.
PF_BUSY	- set if a free frame is being manipulated
PF_FREE	- frame is free
PF_RES	- reserved

A count of present PTEs referencing this page. (16 bits)

A file offset (stored in the VP when there is no PF)

for this page's backing store (either a swapper DF or loader block number). (20 bits)

A count of short-term locks outstanding on the page. (8 bits)

Figure 2. An in-use PF

+-----+									
pVP									
+-----+									
llock		flg		pad		PF ref count			
+-----+									
File offset				pad		slock			
+-----+									
3	2	1	1	1	7	0			
1	3	9	5	1					

A PF is idle if no present PTEs reference it, but the data is still reclaimable (the PF is linked to a valid VP). All idle PFs are kept on a doubly-linked "idle list" from which they may be reclaimed if a PTE that indirectly references the PF is faulted upon, or they may be stolen to use for other data. Since a PF will not be marked idle unless it has a reference count of zero, and since the reference count will never hit zero while there are any outstanding locks on the page, the reference count and lock counts may safely be overlaid by the idle list links.

Figure 3. An idle PF

+-----+									
pVP									
+-----+									
..flink flg back link									
+-----+									
File offset flink...									
+-----+									
3	2	1	1	0					
1	3	9	1						

A PF with a reference count of 0 that is not reclaimable is "free". All free PFs are on a doubly-linked free list. The lay-out of a free PF is the same as an idle PF, except that the file offset field is not currently in use. That data space may be used some day for caching the contents of free pages.

1.3 The PTE

Each committed page in the system may be referenced by one or more Page Table Entries (PTEs). Multiple PTEs referencing the same object may be marked present or not present independently.

Figure 5. A PTE

+-----+ ... +-----+									
frame r u x 0 0 d a c w u r p									
s v d t s w									
+-----+ ... +-----+									
3					1				
1					2				0

frame- if p = 1, PF array index, else VP array index (20 bits)
rs - page referenced is resident (not pageable)
uv - uvirt (direct map of a physical page without a PF)
x - don't care
0 - Intel reserved (2 bits)
d - if p = 1, this is the dirty bit, else the guard bit
a - accessed
cd - cache disable (not currently used)
wt - write transparent (not currently used)
us - user/supervisor access
rw - writeable
p - present

1.4 The DF

Space within the swap file is maintained by a bitmap in which each page in the file is represented by one bit. The index of the bit within the bitmap and the swap file is referred to as the Disk Frame (DF).

2. Page state transitions

2.1 Page allocation

When a swappable or discardable virtual page is committed, a PTE is allocated, marked not present, its frame field is set to the point to a freshly allocated VP, and its reference count is set to 1. The PTE and VP flags are set according to wish of the allocator and the VP frame field is set to a loader block number if the object is initially discarded. When allocating a resident page, the PTE is initially marked as present and both a PF and VP are allocated and cross-linked.

2.2 Attaching to an existing page

When loading a program or library with shared memory, or when shared memory is "given" or "gotten", or when a linear alias is mapped, a page is "attached" to another existing page. An attach is performed by copying the contents of the target PTE to the new PTE, incrementing the VP reference count, and if the PTE is present, incrementing the PF reference count.

2.3 Freeing a page

To free a page, the PTE is zeroed and the VP reference count is decremented. If the PTE was marked present, the PF reference count is decremented as well. If the VP reference count hit zero, the VP and any linked PF are put on their respective free lists. If the PF reference count hit zero and the VP count did not, the PF is put on the PF idle list.

2.4 Idling a page

When short on physical memory, the aging thread is awoken. The aging thread scans all of the page tables in the system and if it encounters a PTE that is present, unaccessed, and pageable, and the associated PF is unlocked, it will idle the page. Idling is performed by marking the PTE not present, putting the VP index in the PTE frame field, and decrementing the PF reference count. If the PF reference count hits zero, the PF is put on the idle list. If the page being idled is marked VP_DIRTY and VP_SOW, the page is converted to swappable.

2.5 Faulting in a page

If the VP for faulting the page is cross-linked to a PF, the PTE is marked present and its frame field is set to the frame value stored in the VP. If there is no cross-linked PF, a PF is allocated off of the PF free or idle list and the page contents are reloaded by swapper or loader, depending on the value of the VP_DISCARDABLE bit.

2.6 Special memory requests

"Special" memory requests are those allocations or locks that require particular physical pages to satisfy. Examples are requests for physical pages below a certain physical address boundary, requests for physically contiguous pages, or requests for memory between linear addresses 4k and 64k whose physical addresses are constrained by a 386 B1 erratum. To meet these requests, the PF array is scanned to find a group of candidate PFs (preferably on the idle list, but all the candidates may be in-use) and these pages are then "stolen". This may involve using the Virtual Memory Manager data structures to enumerate the contexts pointing to the candidates and re-mapping their PTEs elsewhere.

3. Advantages over the old Cruiser page manager

- Page stealing possible for special memory requests.

- Reduced data costs for present and swapped-out pages (PF is 12 bytes smaller, VP is 10 bytes smaller than the SF) that more than offset additional data cost for discarded pages.
- Fewer page states and the common VP make for operations with fewer special cases. That means simpler, smaller, faster code.
- Faulting in a page from the loader no longer requires two hash table look-ups and a bitmap scan (VM is out of the loop).
- No semaphore motion.

Imported interfaces

- ArenaInfo array provided by SYSINIT
 - LDRGetPage()
 - VMAlloc()
 - SELMapVirt()
 - SELAllocGDT()
 - SCHSetPriority2()
 - SELInit()
 - VMPgToHOBMTTE()
 - papTCBSlots
 - SMSwapOut(swapid, vppaddr)
 - SMSwapIn(swapid, vppaddr)
 - SMRequestSwapSpace(npages)
 - npages = SMQuerySwapSpace()
 - SwapSpace dword size of swap file (in pages)
 - SMAllocSF(&pSF)
 - SMFreeSF(pSF)
 - SMIdleSF(pSF)
 - pSF = SMIdleTopSF(swapID)
 - VMEachMap()
- Used by pgDirtyPte to enumerate and discard references to dirty readonly pages, and pgValidateContig and pgMovePF during contiguous lock requests.
- TKBlock(long BlockID, long timeout, int finerruptible) TKRun(long BlockID) TKTCYield()
- Used in various places throughout the Page Manager.

Page state transitions by memory object type

1. EXE/DLL read-only objects
possible types = discardable
possible states = NP.discarded, P.discardable

- Allocate

The pte is marked not-present, discarded, read-only and the mte block number is placed in the pt_frame field.

- NP Fault

The PF cache is searched for a PF matching the mte handle and block number. If one is found, it is attached to the pte. If the PF is not found, one is allocated and the mte block number is copied from the pt_frame field to the pf_blockno field. The PF is placed in the cache, and then the data is loaded. The pte is marked present, pageable and the PF is marked discarded.

- Idle

The pte is marked discarded and the pf_blockno field is copied into the pt_frame field. If the PF reference count is zero, the PF is placed on the idle list.

When running on a 386 without ring 2 read-only page protection, the dirty bit in the pte is checked by the page aging thread during its pte scan. If set, the page frame must be discarded and the process terminated. This is done by marking all ptes referencing the page frame not present and discarded, and setting a force flag in the ptda to terminate the offending process.

2. EXE/DLL instance data
possible types = Commit-on-write, swappable
possible states = NP.discarded, P.discardable, NP.swapped, P.swappable

- Allocate (Attach is done via Allocate)

The pte is marked not-present, read-only, Commit-on-write and the mte block number is placed in the pt_frame field.

- NP Fault

The PF cache is searched for a PF matching the mte handle and block number. If one is found, it is attached to the pte. If the PF is not found, one is allocated, marked Commit-on-write, and the mte block number is copied from the pt_frame field to the pf_blockno field. The PF is placed in the cache, and then the data is loaded. The pte is marked present and pageable.

- Idle

The pte is marked not-present, Commit-on-write, and the pf_blockno field is copied into the pt_frame field. If the PF reference count is zero, the PF is placed on the idle list.

- Write Fault (handle NP fault as above)

If the PF reference count is 1, the PF is removed from the cache. If the reference count is greater than 1, a new PF is allocated, attached to the pte, and the data copied into the PF. The pte is marked writable and pageable, and the PF swappable.

- Idle

A SF is allocated, marked swappable and cross linked to the PF. The SID is placed in the pt_frame field, and the pte is marked swapped and the PF is placed on the idle list.

- NP Fault

If the SF cross link field is not null, the PF is reclaimed. Otherwise, a PF is allocated, marked swappable and the data swapped in. The pte is marked present and pageable.

When running on a 386 without ring 2 read-only page protection, the pte is always marked read/write, and the PF is placed in the PF cache only when its reference count is zero. When the PF is attached to a pte, it is removed from the PF cache, and when it is idled, it is placed back in the PF cache. This implies that only one process can be attached to

a Commit-on-write page at a time, and the page never needs to be copied when a write is detected. Instead of relying on a write fault to detect writes, the dirty bit in the pte is checked when the PF is detached from the pte. If it is set, the page state is translated from Commit-on-write to swappable.

In addition, if an alias (such as a debug alias) is attached to a Commit-on-write page, the page state is immediately changed to swappable. An alternative is to allow all references to a Commit-on-write page share the same frame, at the expense of simplicity.

3. EXE/DLL shared data
possible types = Swap-on-Write, swappable
possible states = NP.discarded, P.discardable, NP.swapped, P.swappable

- a. Attach to discarded pages:

- Allocate

The pte is marked not-present, read-only, Swap-on-Write and the mte block number is placed in the pt_frame field.

- Attach

A SF is allocated, marked Swap-on-Write and the PF cache is searched for a PF matching the mte handle and block number. If found, it is cross linked to the SF. If the PF is not found, the SF_NOXLINK flag is set in the SF, and the mte block number is copied from the pt_frame field to the SF cross link field. The SID is placed in the pt_frame fields of both ptes, and both ptes are marked swapped.

- NP Fault

If the SF SF_NOXLINK flag is not set, the PF is reclaimed. If the SF SF_NOXLINK flag is set, a PF is allocated, marked Swap-on-Write, placed in the PF cache, and the mte block number is copied from the SF cross link field into the pf_blockno field. The PF is cross linked to the SF, and the data is loaded. The pte is marked present and pageable.

- Idle

The pte is marked swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the mte block number is copied from the pf_blockno field into the SF cross link field, and the SF SF_NOXLINK flag is set.

- Write Fault (handle NP fault as above)

The pte is marked writable, and the PF is removed from the PF cache and marked swappable.

- Idle

The pte is marked swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the data is swapped out and the SF cross link field is set to null.

- NP Fault

If the SF cross link field is not null, the PF is reclaimed. Otherwise, a PF is allocated, marked swappable and the data swapped in. The pte is marked present and pageable.

- b. Attach to present, Swap-on-Write pages:

- Allocate

The pte is marked not-present, Swap-on-Write, read-only and the mte block number is placed in the pt_frame field.

- NP Fault

The PF cache is searched for a PF matching the mte handle and block number. If found, it is attached to the pte. If the PF is not found, one is allocated, placed in the PF cache, marked Swap-on-Write, and the mte block number is copied from the pt_frame field to the pf_blockno field. The data is loaded, and the pte is marked present and pageable.

- Attach

The PF cross link is examined to see if a SF has been allocated (in this scenario, a SF has not yet been allocated), and if not, one is allocated and marked Swap-on-Write. The SF is attached to the new pte which is then marked not present, swapped, and read-only.

- Idle

The pte is marked not-present, swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the pf_blockno field is copied to the SF cross linked field, and the SF_SF_NOXLINK flag is set.

- NP Fault

If the SF_SF_NOXLINK flag is not set, the PF is reclaimed. If the SF_SF_NOXLINK flag is set, a PF is allocated, marked Swap-on-Write, placed in the PF cache, and the mte block number is copied from the SF cross link field into the pf_blockno field. The PF is cross linked to the SF, and the data is loaded. The pte is marked present and pageable.

- Write Fault

The pte is marked writable, and the PF is removed from the PF cache and marked swappable.

- Idle

The pte is marked swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the data is swapped out and the SF cross link field is set to null.

- NP Fault

If the SF cross link field is not null, the PF is reclaimed. If the SF cross link field is null, a PF is allocated and cross linked to the SF, and the data swapped in.

c. Attach to swapped pages:

- Allocate

The pte is marked not-present, Swap-on-Write, read-only and the mte block number is placed in the pt_frame field.

- NP Fault

The PF cache is searched for a PF matching the mte handle and block number. If found, it is attached to the pte. If the PF is not found, one is allocated, placed in the PF cache, marked Swap-on-Write, and the mte block number is copied from the pt_frame field to the pf_blockno field. The data is loaded, and the pte is marked present and pageable.

- Idle

The pte is marked not-present and Swap-on-Write, and the pf_blockno field is copied into the pt_frame field.

- Write Fault (handle NP fault as above)

The pte is marked writable, and the PF is removed from the PF cache and marked swappable.

- Idle

A SF is allocated, marked swappable, cross linked to the PF and the SID is placed in the pt_frame field. The pte is marked swapped and the PF is placed on the idle list.

- Attach

The SID is placed in the pt_frame field of the new pte, and the pte is marked not present, swapped and writable.

When running on a 386 without ring 2 read-only page protection, the pte is always marked read/write. Instead of relying on a write fault to detect writes, the dirty bit in the pte is checked when the PF is detached from the pte. If the dirty bit is set, the page state is translated from Swap-on-Write to swappable, and the PF is removed from the PF cache.

If only one process is ever attached to the memory object, the possible transitions are exactly as in EXE/DLL instance data.

4. API private data

possible types = swappable
possible states = NP.Allocate on-Demand, NP.swapped, P.swappable

- Allocate

The pte is marked not-present, Allocate on-Demand and writable.

- NP Fault

A PF is allocated, marked swappable and the pte is marked present and pageable.

- Idle

A SF is allocated, cross linked to the PF, and the SID is placed in the pt_frame field. The pte is marked swapped, and the PF is placed on the idle list.

5. API shared data
possible types = swappable
possible states = NP.Allocate on-Demand, NP.swapped, P.swappable

- a. Attach to Allocate on-Demand pages:

- Alloc

The pte is marked not-present, Allocate on-Demand and writable.

- Attach

A SF is allocated and its id is placed in the pt_frame fields of both ptes. The SF is marked Allocate on-Demand, and both ptes are marked swapped.

- NP Fault

A PF is allocated, marked swappable and cross linked to the SF. The pte is marked present and pageable.

- Idle

The pte is marked swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the data is swapped out and the SF cross link field is set to null.

- b. Attach to present, swappable pages:

- Allocate

The pte is marked not-present, Allocate on-Demand and writable.

- NP Fault

A PF is allocated, marked swappable and cross linked to the SF. The pte is marked present and pageable.

- Attach

A SF is allocated, marked swappable and cross linked to the PF. The SID is placed in the pt_frame fields of the new pte, which is marked not present and swapped.

- Idle

The pte is marked swapped and the SID is placed in the pt_frame field. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the data is swapped out and the SF cross link field is set to null.

- c. Attach to swapped pages:

- Allocate

The pte is marked not-present, Allocate on-Demand and writable.

- NP Fault

A PF is allocated, marked swappable and the pte is marked present and pageable.

- Idle

A SF is allocated and its id is placed in the pt_frame field. The pte is marked not present and swapped. If the PF reference count drops to 0, the PF is placed on the idle list. If the page is removed from the idle list on behalf of an allocation request, the data is swapped out and the SF cross link field is set to null.

- Attach

The new pte is marked not present and swapped, and the SID is placed in its pt_frame field.

If only one process is ever attached to the memory object, the possible transitions are exactly as in API private data.

Selector component

This page is intentionally left blank.

Selector component Architecture

This page is intentionally left blank.

Problem Description/Objectives

This component, previously a functionality included in the VMM, is separated in support of following objectives:

- Support linear address based memory management for applications, bypassing segmented architecture.
- Insure compatibility with existing versions of OS/2, which are based on segmented memory model.
- Improve portability of OS/2 by separating the hardware dependent and independent aspects of memory management.

In OS/2 v 2.0, a flat linear memory model has been adopted to open the way for portability to hardware other than those based on Intel x86 chip. OS/2 v 1.x memory management is based on segmentation and selector based management, a feature that is closely tied to Intel x86 hardware architecture.

OS/2 v 2.0 must be able to run all existing applications by supporting segmented memory model of OS/2 v 1.x. At the same time the flat linear memory model must not depend on segmentation hardware characteristics like selectors and descriptors. This separation is essential to open a pathway for porting OS/2 to hardware architectures other than Intel x86 chip. This concern has led to separation of VMM of OS/2 v 2.0 into two components. Selector Manager will manage all the hardware dependent parts of memory management that must be supported in order to ensure compatibility with existing applications. VMM will implement the flat linear memory model that has been adopted in OS/2 v 2.0 and all the succeeding versions of OS/2. Special attention will be paid to keep data structures of each separate, and interdependence between the two to a minimum.

Solutions/Justification

Segment based memory management of OS/2 v 1.x accepted memory addresses in 16:16 format.(16 bit selector:16 bit offset). Linear address based memory management uses 0:32 (32 bit address) addressing format. Compatibility with current applications require both addressing formats coexist in the system. Furthermore any new 0:32 application may access old 16:16 dynalinks. Therefore OS/2 v 2.0 memory manager must not only support both addressing formats, but also any combination of these. Both the applications and the system must be able to convert addresses back and forth between the two formats with relative ease and efficiency.

Support of both 0:32 and 16:16 addressing formats in the same process requires that there be one to one mapping between the two. In OS/2 v 1.x the maximum amount of memory a process can own is limited by the amount of memory its LDT can map. An LDT can have 8192 descriptors, each mapping up to 64Kb of memory. Therefore the maximum amount of memory an application can have is (8192*64Kb =) 512Megabytes. (This amount excludes system owned memory mapped by GDT, but includes all shared memory allocated by processes in the system.) In OS/2 v 2.0, this limit will be retained for each process in order to have one to one mapping between the two addressing formats. (This limit can be eliminated in a future version). Each LDT descriptor will be mapping a fixed 64Kb of linear address space, and new applications have the option of referring to their memory either in segmented model (loading segment register and using offsets 0 to 64Kb), or in linear addressing model (Using direct 32 bit addresses). Conversion from 16:16 (selector:offset) to 0:32 (linaddress) will be done via the formula:

$$\text{linaddr} = (\text{selector} \gg 3) \ll 16 + \text{offset}$$

The offset in 16:16 format will make up the low 16 bits of the linear address, and the high 13 bits of the selector will make up bits 17 through 29 in the linear address. (The low 3 bits in a selector determine the table bit and ring protection levels, and therefore not indicative of distinct memory addresses.)

Three selectors in GDT will be mapping the whole linear address space accessible by the current process: One as ring 2 conforming code, the second one as ring 3 data, and the third one as ring 2 data. Applications may load these values into segment registers during initialization and use fully linear addressing format, never worrying about segment register loads. Alternatively they can use the fully segmented model compatible with previous versions of OS/2

Architectural Review

To be performed.

Selector component Design

This page is intentionally left blank.

Design Overview

The Selector Manager will be responsible for providing and maintaining the following functionalities:

- Existing 16:16 API Interfaces (DOSALLOCSEG, DOSREALLOCSEG, DOSALLOCHUGE, DOSREALLOCHUGE, DOSFREESEG, DOSCREATECSALIAS,...)
- Existing and new 16:16 DevHlp Interfaces
- Existing 16:16 FSD Interfaces
- Descriptor Management
 - GDT
 - 1. Dynamic Sizing
 - 2. GDT Descriptor Allocation
 - LDT
 - 1. Creation
 - 2. Private Descriptor Allocation
 - 3. Shared Descriptor Allocation
 - Descriptor Initialization and Update
- LDT Context Switching
- Segment Not Present Fault Handling

The ability to convert 0:32 linear addresses to 16:16 segmented addresses forces certain changes in selector managing data structures that are in the scope of Selector Manager in this version:

- The most notable change will be in the HUGEINCR value. The selectors that are HUGEINCR apart must be adjacent to each other in order for 0:32 to 16:16 address conversion to work correctly. Consider a large memory object with HUGEINCR selector map.

We have

```
LinAddr.high = Sel >> 3
```

But we must also have

```
(LinAddr + 64K).high = (Sel + HUGEINCR) >> 3
```

i.e.

```
LinAddr.high + 1 = Sel >> 3 + HUGEINCR >> 3
```

```
1 = HUGEINCR >> 3
```

This requirement blows the three to one ratio of shared to private selectors existing in OS/2 v 1.x. LDTs must be redesigned without this ratio and have selectors that are HUGEINCR apart adjacent to each other, whether they are private or not.

Each LDT will be initialized to have full 64K virtual memory and two valid physical pages (one at beginning, and one at the end). Shared selectors will be allocated from the top, private ones from the bottom. We will be keeping track of how much valid physical memory we have at both ends of LDT, and commit more pages from either end as needed. LDT Bitmap for shared selectors and free private selector link list of OS/2 v 1.x will be eliminated. VMM will allocate linear address space and call Selector Manager to set up the descriptors when needed.

- Huge segments allocated via API DOSALLOCHUGE will be implemented as a single big memory object with selectors that are HUGEINCR apart mapping successive 64K sections of it. This implementation is fully compatible with OS/2 v 1.x, and will also allow new applications to access huge segments from flat linear model.
- Reallocation will be entirely supported within Selector Manager. VMM will not support reallocation functionality, Selector Manager will do this job by using sparse allocation facility provided by VMM. Every segment created will occupy full 64Kb of linear address space in VMM records. The segment will have valid region up to its requested size (rounded up to nearest page boundary), the region beyond its requested size will be allocated as sparse. To VMM, these segments will look like sparse objects. The descriptor limit will be set up to show true requested size of the object. When a reallocation request is received, Selector Manager will call Page Manager to validate or free pages if necessary, and readjust the limit of the corresponding descriptors. The API DOSREALLOCSEG will work on all tiled memory objects with ring 3 data descriptor access bytes (except for huge segments, which will be handled by DOSREALLOCHUGE). The aliases to memory objects can not be reallocated. When an aliased object is reallocated to shrink in size such that part of the alias mapping becomes invalid; debug aliases will be freed, cs aliases will be shrunk in size, other aliases (GlobalToProcess, ProcessToGlobal, VDM) will simply have sparse regions in them. During reallocations to grow, cs aliases will follow the object boundaries up to 64K. Other alias sizes will remain unchanged. Reallocation will be done with page granularity, although the descriptor limit fields will be set to the exact, byte granular size.

- All memory objects in user address space (with the exception of VDM address space) will have to start at a 64K boundary in Linear Address Space (Offset zero in a segment must correspond to the first byte of a memory object, and low 16 bits of a linear address corresponds to the offset in segmented address. Therefore the low sixteen bits must be zero for the starting address of a memory object.).
- All procedures maintaining descriptors will support B(big), G (granularity), D (default) bits in the attribute byte, 20 bit limit and 32 bit address fields. The value of the B/D bit in the descriptor will be determined during the allocation of the descriptor and should never be modified later. The G bit will be set if the size of the linear address map is >1Meg, cleared otherwise.
- Discarding, swapping, demand loading, and allocation on demand will be handled at page level, therefore will be transparent to the Selector Manager. In OS/2 v 2.0, the memory object will be considered present (Present bit in access byte set) even if all of its pages are discarded, swapped out or allocate on demand. Attempts to access memory that is swapped out or not yet allocated will be handled by page faults. The descriptors will always be marked present. It is an Internal Error to get a Segment Not Present Fault in kernel code or on a descriptor with valid (nonzero) access byte.
- Since discarding is done on a per page basis, runtime allocated discardable segments can not be supported. Old applications built around the segmented model can not deal with segments that are partially discarded. In order to keep upward compatibility with old applications, the discardable flag in APIs DOSALLOCSEG and DOSALLOCCHUGE will be ignored and the segments will be allocated as swappable. All DOSLOCKSEG and DOSUNLOCKSEG calls on valid, user accessible data segments will return success without doing anything.

Since LDT pages are swappable in Cruiser, any instructions using LDT selectors (like lsl, lar, mov ds, etc...) may potentially take a page fault and block even if the corresponding segment is present in memory. Device Drivers running with the 'DOES NOT BLOCK' assumption should be aware of this problem and refrain from accessing LDT selectors, using FLAT linear addresses whenever possible.

Page faults taken at interrupt time crash the system, and LDT selectors should never be used at interrupt time. This restriction also exist in the earlier versions of os/2.

In os/2 v1.1 and v1.2, device drivers were able to lock an LDT segment using the DevHelp SegLockDM, and they were assured not to block when they accessed the locked segment. In order to provide the same level of compatibility; whenever SegLockDM DevHelps is called for an LDT segment, the LDT page containing the corresponding descriptor will also be locked; eliminating potential blocks.

LDT pages will not be locked with the new DevHelp VMlock. Potential callers of this new interface for LDT pseudo tiled addresses should not use LDT selectors if they must not block. In this case, simply using (0:32) flat addresses to access memory instead of (16:16) LDT selectors will avoid this problem.

- The new lock handle present compatibility problems for Device Driver Locks. OS/2 v 1.x Device Driver Locks expect to receive a 32 bit lock handle. They also assume that the locked segments are physically continuous in memory. Since device drivers involved in DMA access physical memory directly, bypassing Selector Manager, VMM, and Page Manager, all locks coming through old DevHlp interface (via SegLockDM) will be treated as if the continuity and write flags were set in their lock request. These segments will be locked at physical addresses below 16 Meg. The 96 bit lock handle returned by VMM will be stored in a BMP32 object allocated during system initialization. Its 32 bit linear address will be returned as the 32 bit lock handle. Likewise, SegUnlock will read the 96 bit lock handle from its BMP32 record before passing it along to VMM
- In OS/2 v 1.x PhysToVirt selectors were implemented by reserving a fixed number of selectors during initialization for this purpose and attaching them to physical address ranges whenever the PhysToVirt routine was called. Since writing the physical address to the descriptor did not involve any memory allocation or any other action that could block, these routines were callable at interrupt times. In OS/2 v 2.0, linear instead of physical addresses have to be written to descriptors and these should map to page table entries translating these into physical addresses. If these page table entries are not allocated initially, it will be necessary to allocate them when a PhysToVirt request is received. Allocations may block or fail if due to out of memory conditions. Since this is unacceptable for a routine called at interrupt time, all PhysToVirt selectors will have linear memory and page table entries as well as selectors reserved for them during initialization. The linear address will be written to the descriptor during initialization. The attachment of physical pages to these addresses will be done at the Page Manager level when a PhysToVirt request is received.

Every PhysToVirt selector will have 68Kb of linear address space reserved for it. (64Kb for the maximum size for a 16 bit segment plus one page 4Kb to take into account that the physical address may not start at a page boundary) The starting address of this reserved range will be stored in the descriptor's address field, and should not be changed except for adjustments of less than a page size. When a PhysToVirt request is received, the physical address will be rounded down to nearest page size and the Page Manager will be called to map the physical address to descriptor's reserved page tables. Then the offset of the passed physical address from page boundary will be added to descriptor's address field.

Allocation of GDT selectors via dh_AllocGDTSelector will be handled the same way, i.e. linear memory will also be allocated. This design change has the effect of limiting the range of PhysToVirt and Device Driver reserved GDT selectors to 64Kb (68Kb if page aligned). While this will not break any of the existing code, the new code will also be limited to this size. This DevHlp function, available only during initialization time in previous versions, will be made available during task time in OS/2 v 2.0

- PhysToUVirt selector mapping will consist of allocating LDT selectors, linear address space, and mapping corresponding page table entries to given physical address. Since Page Manager can only put physical addresses that start at a page boundary to

page table entries, and since the LDT selector tiling prevent us from adjusting the base like we do in GDT PhysToVirt selectors; all physical addresses passed to PhysToUVirt must be on a page boundary. Selector Manager will fail requests with physical addresses that are not page aligned.

These are the new DevHlp functions provided by Selector Manager:

- A new, generalized version of the DevHlp interface `dh_PhysToGDTSelector` will be introduced which will enable Device Drivers to specify the access privileges of the GDT selector when mapped to an array of physical addresses. This procedure, `dh_PageListToGDTSel`, will have an extra flags parameter where the caller can specify to have a Ring 0, 2, or 3; code or data mapping in the 16 bit or 32 bit format.
 - `dh_FreeGDTSelector` will free the selectors allocated via `dh_AllocGDTSelector`.
 - `dh_VirtToLin` will convert given virtual address (selector:offset) to corresponding linear (32 bit flat) address.
 - `dh_GetDescInfo` returns information on the contents of the given descriptor.
 - `dh_LinToGDTSelector` will map given GDT selector to to memory mapped by the given linear address and range.
 - In addition to these, macros `SelToFlat` and `FlatToSel` will be provided to convert linear (0:32) addresses to virtual (16:16) and back in the user space.
- The virtual mapping produced by `DevHelp_PhysToUVirt` will be invalidated by calls to `DevHelp_PageListToLin` as well as other `PhysToUVirt` calls.

Data Structures Description

Selector Manager will maintain all descriptors with full 8 byte fields in the 386 format. This includes 32 bit addressing and limits, access and attribute byte properties.

Exported Interfaces

This page is intentionally left blank.

Miscellaneous

SELAddPermission - Increase descriptor access permissions
SELAlloc - Allocate and set up descriptors
SELAllocLDT - Allocate LDT selector
SELAllocVdmSels - allocate consecutive LDT selectors for a DPMI VDM
SELCheckAccess - Check access privileges of the given address range
SELClearPreload - Clear the preload flag of the given descriptor
SELCopy - Copy descriptors from one context to another.
SELCreateVdmLDT - allocate an LDT for a VDM and initialize its free list
SELDestroyVdmLDT - free an LDT for a VDM [//1.5](#)
SELFree - Free descriptor(s) associated with freed memory object
SELFreeLDT - Free LDT selector

SELFreeTaskLDT - Free an entire LDT space.
SELFreeVdmSels - free a group of DPMI LDT selector
SELGetOd - Stabilize segment, then return descriptor and object info.
SELGetSelector - Obtain the selector of the given handle
SELGetSize - Obtain the true size of the object
SELGetVdmInfo - return memory manager info for VDM use
SELInitLDT - Initialize LDT and task's InfoSeg
SELLdrDesc - Map linear address by using the given selector or allocating a new one.
SELMapVirt - map a memory object into the GDT as ring 0 data
SELQueryFreeAlias - Determine whether a cs alias needs to be freed
SELReserveLDT - Reserve selector(s) and virtual memory.
SELSetAccess - Mark LDT descriptors mapping to video buffers
SELSwitchVdmLDT - load new LDT for DPMI VDM task //1.5

SELAllocGDT

SELAllocGDT - Allocate GDT selector

SELAllocGDT allocates GDT selector, growing the GDT if necessary.

```
ENTRY   psel - Address to return selector
RETURN  int   - error code (NO_ERROR if no error)
```

```
SELAllocGDT (sel *psel)
```

PseudoCode:

```
Save all registers
If (GDTFreeList != 0)      ; still selectors on freelist?
  Set desc=[GDTFreeList]
  Set desc.d_limit=0      ; clear limit field
  Set [GDTFreeList] = desc.d_flink
  If ([GDTFreeList] != 0)  ; freelist still not empty?
    set [GDTFreeList].d_blink = 0 ; clear back link field
  Endif
  return(selector)
Endif
; Free list empty, we have to grow GDT
Set nextdesc == [MaxGDTDesc] + size desctab
If (maxdesc >= 64K)      ; GDT already grown to full size?
  return(Carry, ERROR_NOT_ENOUGH_MEMORY)
Endif
Set pg=(nextdesc) >> PAGESHIFT
Call PGAlloc(resident, zeroinit, pg, 1 page) ;allocate one more page
If (error), return (error code)
Set [MaxGDTDesc] = nextdesc + PAGESIZE - size desctab
; set up freelist
Set oldhead= [GDTFreeList]
Set [GDTFreeList] = nextdesc
Do (PAGESIZE/sizeof desctab - 1) times
  set [nextdesc].d_flink = nextdesc+size desctab
  Set nextdesc += size desctab
Set [nextdesc].d_flink = oldhead      ; new block points to old freelist
Reload GDT register to update the limit.
Go back to beginning
```

SELFreeGDT

SELFreeGDT - Free GDT selector

SELFreeGDT puts the given GDT selector back to free list.

```

ENTRY    sel - selector to be freed
RETURN   int - error code (NO_ERROR if no error)
USES     EAX, ECX, EDX, EFlags

SELFreeGDT(SEL sel)

PseudoCode:

Call _selFreeGDTSub

```

_SELVirtToLin

_SELVirtToLin - Convert 16:16 pointer to 0:32

_SELVirtToLin converts 16:16 pointer to 0:32. It handles expand down addresses correctly. No checking is done for LDT selectors, minimal checking for GDT selectors.

```

ENTRY    offset - 32 bit offset
          sel    - selector
          DS = ES = Flat
EXIT     laddr   - 32 bit linear address (-1 if invalid)
USES     EAX, ECX, EDX, Flags
          FS and GS must be preserved (DevHelp requirement)

PseudoCode

If (LDT selector) use SelToLa
Else    read descriptor's address from the GDT
Add the offset returned

```

_SELGetDescInfo

_SELGetDescInfo - Get segment descriptor information

_SELGetDescInfo returns all of the information contained in the specified segment descriptor.
WARNING: This procedure will work for descriptors only after SEL initialization is complete, i.e. after SELInit is called.

```

ENTRY    sel      - selector
          hptda    - ptda handle (required only if LDT selector)
          pseldesc_s - flat address of location to return information
RETURN    int      - error code (NO_ERROR if no error)
          pseldesc_s->desc_cb = size ( = 0 if 4Gb )
          pseldesc_s->desc_laddr = linear address
          pseldesc_s->desc_attr  = attribute (D_UVIRT,D_B, D_G)
          pseldesc_s->desc_acc   = access byte

SELGetDescInfo(SEL sel, VMHOB hptda, struct seldesc_s *pseldesc_s)

Pseudocode:

Mask off RPL and table bits
If (GDT selector)
    If (selector value out of GDT range), return error
    Get GDT Address
Else    /* LDT selector */
    Call selGetLDTAddr
Endif

```

```

Read descriptor fields
If (G bit set) convert limit to byte value
If (expand down segment)
    Call ConvertToExpandUp      /* Convert base and limit to base and size */
Else
    Change limit to size        /* Increment */
Endif
return

```

_SELSetDescInfo

_SELSetDescInfo - Setup segment descriptor information

_SELSetDescInfo optionally sets or updates the limit, base address, access and attribute fields in the specified segment descriptor(s). For a GDT selector, it sets up the descriptor to point to entire size of the memory object. For an LDT selector, it maps consecutive 64K chunks of the object into selectors that are SEL_INCR apart. This routine manages LDT, GDT, shared, and aliased descriptors. VMM alias descriptors are not updated automatically after a memory object limit update. Only the Selector Manager aliases are updated(cs and MemMap aliases).

WARNING: Any change in the first five parameters of this procedure must also be reflected in _SELSetDescAll.

```

ENTRY    (SS:SP)    Scope
          (SS:SP+4)  Access byte
          (SS:SP+5)  Attribute (D_BIG, D_UVIRT only, D_GRAN4K if expand
                     down)
                     If (SELSD_GATE), parameter count (<= 31)
          (SS:SP+8)  Base selector to be set up
          (SS:SP+12) Size of memory object
                     If (SELSD_GATE), offset
          (SS:SP+16) Linear address of base(Needed only if GDT selector)
                     If (SELSD_GATE), selector to store
          (SS:SP+20) Handle of memory object
                     Ignored if SELSD_GATE
          (SS:SP+24) hptda of the selector to be set up
                     If zero, the current task's LDT will be used.
                     Interpreted only if Selector is an LDT selector.
          (SS:SP+28) Oldsize (SELSD_LIMIT and LDT selector only)
RETURN   NONE

```

```

SELSetDescInfo(ulong scope, ushort attr_acc, SEL sel, ulong size,
               ulong laddr, VMHOB handle, VMHOB hptda, ulong oldsize)

```

PseudoCode:

```

Scope parameters
SELSD_GATE Set up descriptor as a (call, interrupt or trap) gate
SELSD_ALL  Set up all fields of the descriptor
SELSD_LIMIT Update only the limit field
SELSD_SHARED Shared memory, update all other contexts (SELSD_LIMIT only)
SELSD_ALIAS Update SELMGR alias selectors (SELSD_LIMIT only)
SELSD_BASEALL Base (LDT) selector maps all (SELSD_ALL and SELSD_LIMIT only)

```

```

#ifdef SELSTRICT
    If (SELSD_ALL || SELSD_GATE)
        If (SELSD_SHARED || SELSD_ALIAS) Panic
    Endif
    IF (SELSD_GATE :: SELSD_BASEALL) Panic
#endif
Convert size to limit and store in stack
Set up helper in stack (sdhgate, sdhLDTall, sdhGDTall, sdhLDTlimit,
sdhGDTlimit)
If (LDT sel), set laddr = selector >> 3 << 16
If (hptda == 0), set hptda = current hptda
If (D_DATA :: D_EXPNDN)
    increment limit
    Call ConvertToExpandDown

```

```

        Save expanddown values on stack
    Endif
    If (SELSD_SHARED or SELSD_ALIAS)
        Call VMEachMap(&selEachDesc)
    Else
        ; limit update for shared LDT selector
        Call selEachDesc
    Endif
    If (selector == DOSGROUP)
        Call Helper(GDT_DOSALIAS)
        Call Helper(GDT_SAS)
    Endif
    return

```

_SELConvertToSelOff

_SELConvertToSelOff - Find the selector and offset, in the LDT

```

                                corresponding to the given linear address.

ENTRY    (SS:SP)    - laddr, linear address
          (SS:SP+4) - rpl bits (-1 if to copy from access byte)
                        0 - Ring 0
                        2 - Ring 2
                        3 - Ring 3
          (SS:SP+8) - hptda, handle to the PTDA
                        0 if GDT sel or current ptda
EXIT     (EAX)      - Selector and Offset
                        (= 0 if error)
                        SELECTOR is high order 16 bits
                        OFFSET  is low  order 16 bits

USES     EAX, ECX, EDX, Flags

ulong_t SELConvertToSelOff(ulong_t laddr, uchar_t rpl, VMHOB hptda)

PseudoCode:

SEL     SELConvertToSelector(ulong_t laddr, uchar_t rpl, VMHOB hptda)

if (return value is zero)
    exit this routine
else
    shift eax 16 bits to left
    add eax to edx (which has the offset from the above call)
endif

return

```

_SELConvertToSelector

_SELConvertToSelector - Find the selector corresponding to the address.

```

SELConvertToSelector converts given linear address to its selector
value and stamps the selector with the given privilege.

ENTRY    (SS:SP)    - laddr, linear address
          (SS:SP+4) - rpl bits (-1 if to copy from access byte)
                        0 - Ring 0
                        2 - Ring 2
                        3 - Ring 3

```

```

                (SS:SP+8)    - hptda
                           0 if GDT sel or current ptda
EXIT   (EAX)              - Selector with desired RPL stamp
                           ( = 0 if error)
                (EDX)              - Offset (passed back to _SELConvertToSelOff)
USES   EAX, ECX, EDX, Flags

SEL     SELConvertToSelector(ulong_t laddr, uchar_t rpl, VMHOB hptda)

PseudoCode:

if (LDT sel)
    LaToSel(laddr)
else
    Call VMGetOd
    read selector
endif
if (rpl == -1)
    read access byte
else
    verify that rpl = (0, 2, 3)
endif
SELStampRPL(sel, access)
return

```

_SELConvertToLaddr

_SELConvertToLaddr - Find the linear address of the selector,

```

                                for the CURRENT PTDA

SELConvertToLaddr converts given selector to its linear address.

ENTRY   (SS:SP+4)    - sel, selector
EXIT    (EAX)        - Linear address
                           ( = -1 if error)
USES    EAX, ECX, EDX, Flags

ulong_t SELConvertToLaddr(SEL)

PseudoCode:

Call SELConvertToLinear (selector, pointer to current PTDA)

return

```

_SELConvertToLinear

_SELConvertToLinear - Find the linear address of the selector

```

                                for the passed PTDA

ENTRY   (SS:SP+4)    - sel, selector
                (SS:SP+8)    - pptda, pointer to target tasks ptda

EXIT    (EAX)        - Linear address
                           ( = -1 if error)
USES    EAX, ECX, EDX, Flags

ulong_t SELConvertToLinear(SEL pPtda)

```

```

PseudoCode:

if (LDT sel)
    SelToLa(sel)
else
    Call GetDescInfo
    if (error), return error
    read address from descriptor info
return

```

_SELSelToLa

_SELSelToLa - converts an arbitrary ldt selector to flat address

```

ENTRY:  sel - selector to convert
        ppta - pointer to target tasks ppta
EXIT:   (eax) = base of selector
USES:   eax,ecx

```

_SELLaToSel

_SELLaToSel - returns selector for a given user space flat address

```

ENTRY:  laddr - linear address to convert
        ppta - pointer to target tasks ppta
EXIT:   (eax) = selector
USES:   eax, ecx

```

_SELSetDescAll

_SELSetDescAll - Set up a descriptor's contents

_SELSetDescAll sets up a descriptor's contents IN ONE CONTEXT ONLY!
 No shared context, no limit, alias, expand down, or gate descriptor supported! For all these combinations, use _SELSetDescInfo.
 WARNING: This procedure is in swappable code
 WARNING: First five parameters of this procedure must be exactly the same as those of _SELSetDescInfo.

```

ENTRY  (SS:SP)      Scope, must be SELSD_ALL
        (SS:SP+4)    Access byte
        (SS:SP+5)    Attribute (D_BIG, D_UVIRT only)
        (SS:SP+8)    Base selector to be set up
        (SS:SP+12)   Size of memory object
        (SS:SP+16)   Linear address of base
        (SS:SP+20)   ppta - Address of ppta

```

_SELInit

_SELInit - Initialize Selector Manager

_SELInit is called by PGInit to update descriptor base addresses after the objects are moved into high end of linear address space. It also reformats 32-bit flat-relative call gates.

ENTRY parena - Pointer to the ArenaInfo array (flat address)
EXIT (EAX) = New flat address of ArenaInfo array
USES EAX, ECX, EDX, Flags

laddr_t _SELInit(struct arena_s *parena_s)

PseudoCode:

```
Save registers for C calling conventions
Set DS = DOSGDTDATA
While (not (parena_s->a_aflags : AF_PEND) ) {
    If (a_oflags : OBJALIAS16 :: a_aflags : AF_LADDR) {
        Set sel = parena_s->a_sel
        set laddr = parena_s->a_laddr
        Set sel descriptor address fields = laddr
        If (sel == GDT selector), save its address
    }
    parena_s += size arena_s
}
Set base address of R0CS and R0DS to 0
; Now all kernel descriptors have been updated. Pop and push all
; segment registers to refresh descriptor caches
Push and then pop  cs, ds, ss, es, fs, gs
Call _TKSetSSBase to update SSBase value
; First update aliases to kernel code segments. Since all segments
; are moving in linear address space once, calculate the mov amount
; and add it to every alias base address
Save new parena_s address
While (a_oflags : OBJALIAS16 :: a_aflags : AF_LADDR)
    parena_s += size arena_s      ; find any valid map
Set basemove = laddr - paddr
Set tabOffst = offset INITCODE:GDTInitTab
While (tabOffst != 0) {
    Set sel= [tabOffst]
    Read address fields from sel descriptor
    Add basemove to address
    Store the new address back into the descriptor
    If (sel == GDT_IDT)
        Update IDTDesc
    Endif
    Set tabOffst += 4              ; get next selector
}
Set up GDTDesc                    ; new GDT address
Load  GDTR
Load  IDTR
restore saved registers
call _selFixCallGates to reformat 32-bit flat-relative call gates
return new arena info address
```

_SELReinit

_SELReinit - Complete initialization of Selector Manager

_SELReinit is called by VMInit to move GDT into its final address

and initialize lockhandle BMP32 records object.

```
ENTRY    NONE
EXIT     NONE
USES     EAX, ECX, EDX, Flags
```

PseudoCode:

```
Call PGAllocCRPTE - allocate user page tables
Call VMAlloc(sparse, 64K, system area, resident) - GDT
Map GDT, read GDT descriptor size
Call PGAttach      ; attach GDT pages to newly allocated high memory
If (size : PAGEMASK) ; size wasn't on a page boundary
    Panic          ; size must be on a page boundary
Endif
Set up DOSGDTDATA descriptor
Load GDT register
Set _SELGDTLaddr = new flat address of GDT
Call VMFreeMem(old GDT pointer)
Call _SELlockHandleInit
return
```

_SELEndInit

_SELEndInit - Free OS2LDR data segment (GDT_OS2LDR)

_SELEndInit is called by Sysinit to free the OS2LDR data segment
It also sets Device Drivers FLAT code segment to nonconforming.

```
ENTRY    NONE
EXIT     NONE
USES     EAX, ECX, EDX, Flags
```

PseudoCode:

```
Call _SELGetOd(GDT_OS2LDR)
Call _VMFreeMem()
;;;;;seiel: Panic  <'SELEndInit: SELObjectInfo'>
```

SELInitRM

SELInitRM - Initialize GDT

SELInitRM initializes system's GDT. Uses a parameter
table to initialize the static entries in the GDT.

```
ENTRY    (ES:DI) = ArenaInfo array
EXIT     (ES)    = DOSINITDATA paragraph address
          (EAX)   = Base address in R0CS and R0DS
          (EBX)   = Offset of ArenaInfo w.r.t. R0DS, R0CS
                  right after we switch to protmode
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags
```

Algorithm:

```
Find and save the physical addresses of dosmte,
    initdataseg, highdataseg, and GDT.
Complete static descriptor initialization by adding the physical
address of the group to the alias descriptors for DFTSS, DFSTACK,
GDT, IDT, SAS, TSS, INTSTACK and INFOSEG.
Find the first element in the arena info array with a high virtual
address. Calculate (paddr - vaddr) and store this value for
```

```

        later use as base address of R0CS and R0DS
Calculate the flat address of ArenaInfo array in Protected mode, twice;
once for before paging is initialized, once after PG initialized
and kernel segments moved into high memory.
Call GDTMapInit to walk the arena info array to map every segment
that requires mapping. Store (paddr - vaddr) as calculated above in
R0CS and R0DS as base address. For other selectors, store
given physical addresses.
Set DOSGROUP and DOSINITDATA to ring 3.
Initialize DevHlp call gate
Initialize GDTDesc and IDTDesc (for loading into GDTR and IDTR)
Call GDAllocInit to initialize the GDT free selector link list.

```

SELReinitDPL

SELReinitDPL - remap DPLs at the end of init time.

This routine changes a bunch of descriptors from their init time state of DPL 3 to their final resting place at DPL 0.

```

ENTRY:  none

EXIT:   none

USES:   EAX,EBX,DS,ES,FS

```

SegLockDM

SegLockDM - Lock Memory Segment

SegLockDM is called by device drivers, at task time, via the DevHlp interface, to lock a memory segment. If the segment is unavailable (swapped out) the caller specifies whether SegLock is to block until the segment is available and locked, or to return immediately.

This procedure can only lock segments up to 64K in size.

Called ONLY from the DevHlp dispatcher in KERNEL MODE and at initialization time. Causes a devhlp lock.

This procedure has been optimized to handle short term prot. mode locks most efficiently.

```

ENTRY  (AX) = Segment or Selector (mode dependent)
       (BH) = Duration (bit field):
           ML_LONG - long term duration (else short term)
           ML_HIMEM - Fix segment permanently (init time only)
           ML_VERIFY - Don't lock the segment
           ML_NONCONTIG - Lock segment discontinuously
       (BL) = Blockflag:
           MW_BVALID0 - block until segment locked
           MW_TEST - return if segment not immediately available
EXIT   Carry Flag clear on success
       (AX:BX) = lock handle
       Carry Flag set on error
       (AX) = error code
USES   EAX, EBX, Flags

```

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

```
(These locks should be contiguous, below 16 Meg, and with write request)
Add ML_DEVHLP to lock flags
Call SegLock
If (error), return error code
Else    return lock handle
Endif
...
```

SegUnlockDM

SegUnlockDM - Unlock Memory Segment

SegUnlockDM is called by 16 bit device drivers, at task time, via the DevHlp interface, to unlock a memory segment.

Called ONLY from the DevHlp dispatcher in KERNEL MODE and at initialization time. Undoes a devhlp lock.

```
ENTRY    (AX:BX) - Lock handle
EXIT     Carry Flag clear on success
          Carry Flag set on error
USES     Flags
```

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

```
If (lockhandle == SELNOPOWNER), return
Call SegUnlock
return
```

SegAlloc

SegAlloc - Allocate a 16 bit segment, handle and selector.

SegAlloc allocates an LDT or GDT selector, a memory segment handle and possibly physical memory. This routine is called to set up the first access to a 16 bit segment that does not yet exist.

NOTE:SegAlloc only allocates GDT or private LDT, Ring 0 or 3 writable data segment. For any other combination of memory, use VMAllocMem

```
ENTRY    ((SS:SP)) = Selector - the selector to be used
          Required only if SA_SPECIFIC set in SFlags,
          caller cleans stack if necessary.
(AX:BX) = Size - the segment's size.
          zero is an invalid size
(CL) = SFlags - selector allocation flags:
          SA_GDT - GDT selector if clear
          SA_LDT - LDT selector if set
          SA_SPECIFIC - use specific selector at ((SS:SP)) (SA_LDT)
          SA_ALLOCATED - selector at ((SS:SP)) already allocated
(CH) = AFlags - access flags
          D_PRIV field set to D_DPL[0/3]
(DX) = TaskPTDA - handle to the PTDA
(SI) = HFlags - Handle/memory allocation flags
```

```

        HF_FIXED set if fixed memory needed
        HF_MOVABLE if movable memory needed
        HF_SWAPPABLE set if swappable
        HF_LOWMEM set if <1M memory needed (defaults to fixed)
        HF_ZEROINIT set if memory must be zero initialized
        HF_HIMEM set if high memory needed
        HF_NONCONTIG set if non-contiguous fixed memory needed.
(DI) = OwnerID - the owner of the segment
EXIT    Carry Flag clear on success
        (AX) = Handle
        (BX) = Selector
        Carry Flag set on error
        (AX) = VM error code
        (BX) = sub-component error code
USES    EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags

PseudoCode:

Rearrange input parameters
Call _SegAlloc
Return (handle, selector)

```

SegRealloc

SegRealloc - reallocate a system-managed, 16 bit segment

This function is called to realloc non-huge non-csaliased non-discardable non-PhysToUVirt LDT and GDT segments. Attempts to realloc other segments on behalf of kernel callers will cause InternalErrors; ifs callers will be returned an error. LDT segments must be mapped through the current task's LDT.

```

ENTRY    (AX:BX) = Size - requested size
        (DX)    = ptda handle (LDT segments only)
        (DI)    = Selector - LDT or GDT selector
EXIT     Carry clear on success
        Carry set on error
        (AX) = ERROR_INVALID_ACCESS
        OR
        (AX) = 0
        (BX) = sub-component error code
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EFlags

PseudoCode:

Call SELSegRealloc
return

```

SegFree

SegFree - free a system-managed segment

This function is called to free non-huge non-shared non-csaliased non-PhysToUVirt LDT and GDT segments. Attempts to free other segments on behalf of kernel callers will cause InternalErrors; ifs callers will be returned an error. LDT segments must be mapped through the current task's LDT.

```

ENTRY    (AX) = Selector
        (CX) = 1 (ifs_SegFree only)
EXIT     Carry clear on success
        Carry set on error

```

```
USES    EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags
```

```
PseudoCode:
```

```
Call SELSegFree  
return
```

SegFree2

SegFree2 - free a system-managed segment

This procedure is called from a 32-bit segment

This function is called to free non-huge non-shared non-csaliased non-PhysToUVirt LDT and GDT segments. Attempts to free other segments on behalf of kernel callers will cause InternalErrors; ifs callers will be returned an error. LDT segments must be mapped through the current task's LDT.

```
ENTRY    (AX) = Selector  
         (CX) = 1 (ifs_SegFree only)  
EXIT     Carry clear on success  
         Carry set on error  
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags
```

```
PseudoCode:
```

```
Call SELSegFree  
return
```

SegLock

SegLock - Lock Memory Segment

SegLock is called by the kernel or by device drivers at task time, via the DevHlp interface, to lock a memory segment. If the segment is unavailable (swapped out) the caller specifies whether SegLock is to block until the segment is available and locked, or to return immediately.

```
ENTRY    (AX) = Selector  
         (BH) = Duration(bit field):  
             ML_LONG - long term duration (else short term)  
             ML_DEVHLP - device driver lock request  
             ML_VERIFY - verify lock. Don't make memory resident.  
             ML_NONCONTIG - Lock memory discontinuous  
         (BL) = Blockflag:  
             MW_BVALID - block until segment locked  
             MW_TEST - return if segment not immediately available  
EXIT     Carry Flag clear on success  
         (AX:BX) = lock handle  
             If call originated in protected mode, both halves  
               of the lock handle are guaranteed to be non-zero.  
         Carry Flag set on error  
         (AX) = (BX) = error code  
USES     EAX, EBX, ECX, EDX, ESI, DS, ES, Flags
```

```
PseudoCode:
```

```
If (LDT selector and NOBLOCK) return ERROR_NOBLOCK  
Call SelGetDescInfo      ; get descriptor info for selector  
If (error || not D_SEG), return ERROR_ACCESS_DENIED
```

```

Translate SegLock flags to VMLockMem flags, adding contiguity and
write access flags
If (LDT selector)
    SegLockSub(LDT page)
    If (error), return
Endif
Call SegLockSub                ; lock the segment and write
                                ; 96-bit lock handle to BMP32 record
If (error)
    If (LDT selector)
        Call SegUnlockSub
    Endif
    return (error code)
Endif
Link LDT lockhandle to segment lock handle record
return (ax:bx = compressed lock handle = BMP32 record handle: object
        handle contained in lockhandle)

```

SegUnlock

SegUnlock - Unlock Memory Segment

SegUnlock is called by the kernel or by device drivers at task time, via the DevHlp interface, to unlock a memory segment.

```

SegUnlock:
    ENTRY (AX:BX) - Compressed Lock handle
                    (AX) = Handle of BMP32 record for 96-bit lock handle
                    (BX) = Object Handle located in 96-bit lock handle
    EXIT    Carry Flag clear on success
            Carry Flag set on error
            (AX) = (BX) = error code
USES      EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags

PseudoCode:

call BMPHandleMap to map BMP32 record handle in (AX) to pointer
to 96-bit lock handle
If ((BX) does not match Object Handle in 96-bit lock handle)
    return ERROR_INVALID_PARAMETER in (BX), 0 in (AX), carry set
Call SegUnlockSub
If (error)
    return ERROR_INVALID_PARAMETER in (BX), 0 in (AX), carry set
Endif
If (a corresponding LDT lock exists)
    call BMPHandleMap to map the BMP32 handle to pointer
    Call SegUnlockSub to unlock LDT page
    If (error), InternalError
Endif
return NO_ERROR in (BX), (AX) and carry clear

```

SegGetInfo

SegGetInfo - Return handle size, and state of passed sel in each case

This function is called to convert a selector into a handle and return information on the segment.

```

ENTRY    (AX) = selector
          (DX) = ptda handle (only required for LDT selector)

```

```

EXIT    Carry clear on success
        (AX:BX) = size of segment
        (CH)    = attribute byte
        (CL)    = access byte
        (DX)    = owner
        (SI)    = handle
        (ES:DI) = FLAT:0 (catch those trying to write to old
                          handle data structure)
        Carry set on error
        (AX) = VM error code
        (BX) = sub-component error code
        ERROR_DIRECT_ACCESS_HANDLE
            (CH) = attribute byte
            (CL) = access byte
            Selector is a PhysToUVirt selector
        else
            Selector is the GDT null selector
            Selector is too large for LDT/GDT
            Selector does not refer to a segment

USES    EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags

PseudoCode:

Call SELGetOd
if (error), return error code
return output parameters

```

SegMapVirt

SegMapVirt - map a real mode segment into the GDT as ring 0 data.

SegMapVirt optionally allocates a GDT selector, maps the associated descriptor as ring 0 r/w data.
 Selector RPL is unaffected, 0 if GDTAlloc called.

```

ENTRY    (AX) = paragraph to map (linear address to map)
        (BX) = 0 or GDT selector
        (CX) = size (0 = 64k)
EXIT     (AX) = selector
USES     EAX, EBX, ECX, EDX, ESI, DS, ES, Flags

```

SELSegRealloc

SELSegRealloc - reallocate a system-managed, 16 bit segment

This function is called to realloc non-huge non-csaliased non-discardable non-PhysToUVirt LDT and GDT segments. Attempts to realloc other segments on behalf of kernel callers will cause InternalErrors; ifs callers will be returned an error. LDT segments must be mapped through the current task's LDT.

```

ENTRY    (AX:BX) = Size - requested size
        (CX)    = 1 (ifs_SegRealloc), 0 otherwise
        (DX)    = ptda handle (LDT segments only)
        (DI)    = Selector - LDT or GDT selector
        DS = ES = FLAT
EXIT     Carry clear on success
        (AX) = NO_ERROR
        Carry set on error
        (AX) = (BX) = error code
USES     EAX, EBX, ECX, EDX, ESI, EDI, EFlags

```

```

PseudoCode:

Call _SELGetOd          ; Find linear address
If (error), Goto Error2Exit
If (not segment desc. || D_UVIRT), Goto Error2Exit
If (HF_LALIAS)
    Call VMSELAlias(export) ; export laddr of alias records
    If (CS aliased), Goto Error1Exit(err= ERROR_ACCESS_DENIED)
Endif
If (ifs caller)
    If (HF_SHARED), Goto Error1Exit(err=ERROR_ACCESS_DENIED)
    If (GDT selector)
        If (not FSD owner )
            Goto Error1Exit(err= ERROR_ACCESS_DENIED)
        Endif
    Else
        ; LDT selector
        If (owner != current ptda) Goto Error1Exit(err=ERROR_ACCESS_DENIED)
    Endif
Endif
Call ReallocSub
Error1Exit: Call VMClearSem
Error2Exit: If (ifs)
    return (error code)
Else
    InternalError
Endif
return

```

SELSegFree

SELSegFree - free a system-managed segment

This function is called to free non-huge non-shared non-csaliased non-PhysToUVirt LDT and GDT segments. Attempts to free other segments on behalf of kernel callers will cause InternalErrors; ifs callers will be returned an error. LDT segments must be mapped through the current task's LDT.

```

ENTRY    (AX) = Selector
         (CX) = 1 (ifs_SegFree only), 0 otherwise
         (DS) = (ES) = FLAT
EXIT     Carry clear on success
         Carry set on error
USES     EAX, EBX, ECX, EDX, ESI, EDI, EFlags

```

```

PseudoCode:

Call SelGetOd
If (error || OB_LALIAS || OB_SHARED || OB_UVIRT)
    If (ifs), return ERROR_ACCESS_DENIED
Else
    InternalError
Endif
Rearrange input parameters
Call VMFreeMem
return

```

w_AllocSeg

w_AllocSeg - allocate a data segment

*** w_AllocProtSeg - allocate a data segment in protected memory if possible

w_AllocSeg allocates a private or unnamed shared data segment in the 286 chip, 16 bit format.
w_AllocProtSeg is identical to w_AllocSeg except that a GIVESEG or GETSEG segment will be allocated from the protected region of the shared arena.

ENTRY (AX) = allocation flag bits:
0: private segment
1: GIVESEG segment
2: GETSEG segment
4: discardable segment
8: shrinkable segment
(BX) = size in bytes (0 = 64K)
EXIT Carry clear
(AX) = selector
Carry set
(AX) = error code
USES EAX, EBX, ECX, EDX, EDI, ESI, DS, ES

PseudoCode:

Set or clear VMAC_PROT flag
Call selAllocSeg

w_ReallocSeg

w_ReallocSeg - realloc a segment

This function is called to realloc 16 bit unshared segments and shared segments. It will fail on CS aliases and huge segments.

ENTRY (AX) = selector
(BX) = size in bytes (0 = 64K)
EXIT Carry clear on success
Carry set on error
(AX) = error code
USES EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags

Pseudocode:

Call selReallocSeg
return

w_FreeSeg

w_FreeSeg - free a segment

This function is called to free unshared segments, shared segments, huge segments, and CS aliases, and local aliases. When called with an alias selector, it will simply unmap the selector from the segment.

WARNING: This procedure must not block between the return from SELGetOd call to VMFreeMemOd call. Otherwise the information in object buffer (vmoi_s) may become invalid.

```

ENTRY    (AX) = selector
         (BX) = flags (No longer used)
         (DX) = ptda handle (No longer used, current ptda assumed)
EXIT     Carry set
         (AX) = error code
         Carry clear
         no error
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags

PseudoCode

If (not LDT selector) return(ERROR_ACCESS_DENIED)
Call    SELGetOd(don't take sem)
If (external request :: Ring 0 sel) return (ERROR_ACCESS_DENIED)
If (UVIRT : Ring 0 || infoseg)
    return (ERROR_ACCESS_DENIED)
Endif
Call VMFreeMemOd
return

```

w_GiveSeg

w_GiveSeg - Give access to a sharable segment to another task.

**** w_GiveSeg2** - Give access to a sharable segment to another task.

This function is called to allow another task access to a sharable segment.

```

ENTRY    (AX) = Selector
         (BX) = ProcessID          (w_GiveSeg)
         (DX) = PTDA handle        (w_GiveSeg2)
EXIT     Carry clear on success
         (AX) = selector
         Carry set on error
         (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags

PseudoCode:

Call selGiveSeg
return

```

w_GetSeg

w_GetSeg - Get access to a sharable segment.

This function is called to allow this task access to other task's sharable segments.

```

ENTRY    (AX) = Selector
EXIT     Carry clear on success
         (AX) = selector
         Carry set on error
         (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags

PseudoCode:

Call selGetSeg

```

```
return
```

w_LockSeg

w_LockSeg - Discard-lock a segment.

This function is called to lock a segment to keep it from being discarded.

```
ENTRY    (AX) = Selector
EXIT     Carry clear on success
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags
```

PseudoCode:

```
call selLockSeg
```

w_UnlockSeg

w_UnlockSeg - Discard-unlock a segment.

This function is called to lock a segment to keep it from being discarded.

```
ENTRY    (AX) = Selector
EXIT     Carry clear on success
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES, EFlags
```

PseudoCode:

```
call selUnlockSeg
```

w_AllocHuge

w_AllocHuge - Allocate a huge memory segment

*** w_AllocProtHuge - Allocate a huge memory segment in protected memory

w_AllocHuge allocates a huge memory segment. We allocate a sparse object for the maximum size, and then validate the initial requested size. Selectors are set up to show the requested size. w_AllocProtHuge does the same except the memory is allocated from the protected region of the shared arena if shared memory is requested.

```
ENTRY    (AX) = NumSeg - number of 64k segments to allocate.
          (BX) = Size - size of the last segment to allocate.
```

```

(CX) = MaxNumSeg - the maximum number of segments ever
      to be required by a w_ReallocHuge call (or 0).
(DX) = allocation flag bits:
      0: private segment
      1: GIVESEG segment
      2: GETSEG segment
      4: discardable segment
      8: shrinkable segment
EXIT  Carry Flag clear
      (AX) = Selector - the base selector of the newly
           allocated huge segment.

      Carry Flag set
      (AX) = error code
USES  EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EFlags

Pseudocode:

Call selAllocHuge
return

```

w_ReallocHuge

w_ReallocHuge - Reallocate a huge memory segment

```

w_ReallocHuge changes the size of a huge memory segment.

ENTRY  (AX) = NumSeg - number of 64k segments desired.
      (BX) = Size - the size of the last segment desired.
      (CX) = Selector - huge base selector.
EXIT  Carry Flag clear
      NONE
      Carry Flag set
      (AX) = error code
USES  AX, BX, CX, DX, SI, DI, DS, ES, Flags

PseudoCode:

Call selReallocHuge
return

```

w_GetHugeShift

w_GetHugeShift - Get huge model selector increment shift count

```

w_GetHugeShift returns the huge model selector increment shift
count. The huge selector increment is computed by shifting
one left by the w_GetHugeShift return value. This routine
is called by the DosGetHugeShift API. No error is possible.

ENTRY  NONE
EXIT  Carry Flag clear
      (AX) = Increment Shift Count
USES  AX

```

w_CreateCSAlias

w_CreateCSAlias - Create 16 bit code selector to

access a data segment

This function allocates a selector in the LDT, and sets it up so that it can be used as 16 bit code selector for the given data segment. The value of the 16 bit code selector is returned.

```
ENTRY    (AX) = Data Selector to be aliased.
EXIT     Carry clear on success
          (AX) = 16 bit code Selector
          Carry set on error
          (AX) = Error code
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EFlags
```

PseudoCode:

```
Call selCreateCSAlias
return
```

w_MemAvail

w_MemAvail - return size of free memory

This function returns the amount of free memory in the system at the time of the call. This is only a snapshot, and can be expected to change.

```
ENTRY    None
EXIT     DX&AX = size in bytes of free memory
USES     EAX, EBX, ECX, EDX, ESI, EDI, EBP, DS, ES
```

w_SizeSeg

w_SizeSeg - return size of a segment.

This function returns the size of an allocated segment.

```
ENTRY    (AX) = Selector
EXIT     Carry clear on success
          (DX&AX) = size of segment in bytes
          (CX) = max number of selectors (>1 if huge)
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, DS, ES
```

MemMapAlias

MemMapAlias - Map an alias for the passed handle's segment.

MemMapAlias returns a Selector that points to the passed handle's memory segment as ring 0 writable data.

```
ENTRY    (SI) = Handle - handle of segment to be mapped.
EXIT     Carry clear
          (AX) = Selector/Segment
          Carry set
          (AX) = Error code
USES     EAX, EFlags
```

PseudoCode:

```
Save all registers
Call SELMemMapAlias
return
```

MemUnmapAlias

MemUnmapAlias - Unmap an alias mapped through the passed selector.

MemUnmapAlias unmaps an alias mapped by MemMapAlias.

```
ENTRY    (AX) = Selector - selector to be invalidated.
EXIT     NONE
USES     EAX, ESI, ES, Flags, and DS if (DS) = (AX)
```

PseudoCode:

```
Call SELObjectInfo          ; get info on object
if (not OB_LALIAS) InternalError
Call VMSELAlias(MEMMAP)
If (error), InternalError
Decrement ref. count
If ref count == 0,
    Call VMFreeMem
Endif
Call VMClearSem
return
```

GDTAlloc

GDTAlloc - Allocate a permanent GDT selector.

Calls SELAllocGDT to allocate GDT selector from the GDT.

```
ENTRY    NONE
EXIT     C clear if success
          (AX) = GDT Selector
          C set if error
          (AX) = error code
USES     EAX, Flags
```

GDTFree

GDTFree - Free a GDT Selector

GDTFree call `_SELFreeGDT` to free the associated selector.

ENTRY (AX) = Selector - GDT selector to be freed.
EXIT NONE
USES AX, ES, Flags, and DS if DS == AX

Pseudo code:

Call `SELFreeGDT`

SetDescInfo

SetDescInfo - Setup segment descriptor information

SetDescInfo sets the limit, base address, access and attribute fields in the specified segment descriptor(s).

This routine basically, rearranges parameters and calls `_SELSetDescInfo`.

WARNING: SetDescInfo does not set up gate descriptors correctly
Use `_SELSetDescInfo` to set up gate descriptors.
This procedure is there for support of old code only. It doesn't handle aliased segments correctly.

Most parameters are passed on the stack; SetDescInfo cleans the stack.
Before call or `CALLFAR`:

((SS:SP)) Size - the 32 bit segment size, not the limit.
Converted to a limit before editing descriptors.
((SS:SP+4)) Address - the current 32 bit physical address
((SS:SP+8)) Access - the access rights for the segment.
((SS:SP+9)) Scope - see `virtsym.inc`: "SetDescInfo flag arguments"
Only `SD_ALL` and/or `SD_UVIRT` is valid
((SS:SP+10)) Selector - for the descriptor to be setup.
((SS:SP+12)) Ptda - owner of the LDT to be edited/examined.
If zero, the current task's LDT will be used.
Interpreted only if Selector is an LDT selector.
EXIT Carry Flag clear on success
USES EAX, ECX, EDX, DS, ES, Flags

GetDescInfo

GetDescInfo - Get segment descriptor information

GetDescInfo returns the limit field, base address, access and attribute fields from the specified segment descriptor.

ENTRY (DI) = Selector - the descriptor to be examined.
(DX) = TaskPTDA - handle to the PTDA in who's LDT the

```

        descriptor information will be found. If
        TaskPTDA is zero, the current task's LDT will
        be used. Required only if Selector is an LDT
        selector.
EXIT    Carry Flag clear on success
        (AX:BX) = Address - the current 32 bit linear
        address of the memory segment.
        (CX:DX) = Size - a 32 bit length for the segment,
        not the limit. The Size value will be
        converted from the limit by this routine
        when extracted from the descriptor. (0 if size = 4Gb)
        (SI)    = Attr/Access - attr/access bytes.
        Carry Flag set on error
USES    EAX, EBX, ECX, EDX, DI, SI, DS, ES

```

HandleGetSelector

HandleGetSelector - Translate a handle to its selector

HandleGetSelector returns the selector used by the passed handle.
 WARNING: This code is obsolete, new callers should use _SELGetSelector.

```

ENTRY    (SI) = Handle - handle to dereference
EXIT     Carry clear on success
        (AX) = Selector
        Carry set on error
        (AX) = 0 (to cause caller GP fault)
USES     EAX, ES, Flags

```

Pseudo code:
 return(Handle.selector)

_SELMemMapAlias

_SELMemMapAlias - Map an alias for the passed handle's segment.

_SELMemMapAlias returns a Selector that points
 to the passed handle's memory segment as ring 0 writable data.

```

ENTRY    (SS:ESP) = Handle - handle of segment to be mapped.
        DS = ES = FLAT
EXIT     (ECX) = Error code
        (AX) = Selector (if ECX = no error)
USES     EAX, ECX, EDX, EFlags

```

PseudoCode:

```

Call VMObjHandleInfo ; get laddr, context
Call VMGetMemInfo(takesem)
If (error), InternalError
If (OB_LALIAS)
    Call VMSelAlias ; export MEMMAP alias record
    if (found)
        increment reference count
        go to VMClearSem
    endif
Endif
Call VMMapAlias(reserve system memory, ring 0 read only data)
Call VMSelAlias ; export alias record's address
Set ref count = 1, sel_seldata = selector, sel_selcode = 0
Call VMClearSem
restore all registers

```

```
return
```

_SELMemUnmapAlias

_SELMemUnmapAlias - Unmap an alias mapped through the passed selector.

```
_SELMemUnmapAlias unmaps an alias mapped by MemMapAlias.

ENTRY    (SS:SP) = Selector - selector to be invalidated.
EXIT     NONE
USES     EAX, ECX, EDX, EFlags

PseudoCode:

Call SELObjectInfo          ; get info on object
if (not OB_LALIAS) InternalError
Call VMSELAlias(MEMMAP)
If (error), InternalError
Decrement ref. count
If ref count == 0,
    Call VMFreeMem
Endif
Call VMClearSem
return
```

SELLoadSegment

SELLoadSegment - load a not present segment

```
Entrypoint from TASK\TRAP.ASM

This routine loads a not present segment in response to a
segment fault or segment preload.

ENTRY:   ES = DS = Flat
         EAX = mte handle to load segment from
         EDX = LDT selector to load (ldt and rpl bits undefined)
         EDI = pointer descriptor

EXIT:    EAX = NO_ERROR if success, else error code

USES:    eax, ecx, edx, eflags
```

DevhlpFreeGDTSel

DevhlpFreeGDTSel - Free GDT selectors

DevhlpFreeGDTSel frees the GDT selectors allocated via
DevhlpAllocGDTSel.
This routine can not be called at interrupt time.

```

ENTRY    (AX)      = GDT selector to be freed
EXIT     Carry clear if success
          Carry Flag set on failure
          (AX) = ERROR_INVALID_PARAMETER
          ERROR_ACCESS_DENIED
USES     EAX, Flags (Also DS, ES, FS and GS if they are equal to
          selector being freed.)

```

PseudoCode:

```

If LDT selector, return ERROR_INVALID_PARAMETER
Get access rights
If (invalid), return ERROR_INVALID_PARAMETER
Read address from descriptor
Call _VMGetOd
If (owner != PTOGDTOWNER || not UVIRT)
    return ERROR_ACCESS_DENIED
Endif
Call VMFreeMem
return

```

VirtToPhys

VirtToPhys - convert virtual to physical address (protected mode only)

*** VirtToPhysDM - convert virtual to physical address (dual mode)

When in real mode, VirtToPhys converts a segment-offset pair to a 32-bit physical address. When in protected mode, it converts a selector-offset pair to a 32-bit physical address.

Calling this routine is allowed at interrupt time, but only on GDT selectors.

Calling in real mode can only happen during system initialization. Callable in user mode and kernel mode.

Algorithm:

- if real mode convert segment:offset to physical address
- If protected mode get linear address from descriptor, and call PGPageToPhys

```

ENTRY    DS:SI = virtual address:
          segment:offset (real mode) or
          selector:offset (protected mode)

```

```

EXIT     'C' Clear if no error
          AX:BX = 32-bit physical address
          'C' Set if virtual address error

```

```

USES     EAX, EBX, Flags

```

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

```

If (realmode)
    Convert segment:offset to physical address
Else
    ; protected mode
Save all registers
If (LDT selector)
    Call GetLDTAddr(Current ptda)
Else
    Get GDT address
Endif
Convert selector to offset
Add selector offset to LDT address
Read d_loaddr, d_hiaddr, and d_extaddr fields to eax
Add value of si to eax

```

```

Call VMLinToPhys
If (error), return (error code)
Save address
Restore registers
Put physical address in ax:BX
return

```

dh_PhysToUVirt

dh_PhysToUVirt - top level worker for DevHlp_PhysToUVirt

```

ENTRY:  (AX:BX) = 32-bit physical address
        On dh=2 AX is LDT selector to free
        (CX)    = size of segment, 0 == 64k
        (DH)    = 0: make segment readable code
                1: make segment writable data
                2: free selector
                3: make segment readable IOPL code
                4: make segment read/write IOPL data
                5: make segment read/write data, tag selectors as PVB
        (SI)    = 0 (used only if dh=5)
EXIT:   Carry Flag clear on success
        ES:BX = valid virtual address
        Carry Flag set on error

USES:   BX, ES, Flags

        FS and GS must be preserved (DevHelp requirement)

```

AllocPhysDM

AllocPhysDM - Allocate Fixed Block of Physical Memory

AllocPhysDM is used by device drivers to allocate a block of fixed memory. This routine can not be called in interrupt time.

```

ENTRY   (AX:BX) = 32-bit block size in bytes
        (DH)    = Relative position to 1 MEG
                = 0 above 1 MEG
                = 1 below 1 MEG
EXIT    Carry clear on success
        (AX:BX) = 32-bit physical address
        Carry set on error
        (AX) = error code
USES    All other registers preserved

        FS and GS must be preserved (DevHelp requirement)

```

PseudoCode:

```

#ifdef SELSTRICT
    If (real mode), InternalError
#endif
Call AllocPhys
return

```

FreePhysDM

FreePhysDM - Allocate Fixed Block of Physical Memory

FreePhysDM is used to release memory allocated by the AllocPhys call.
In current version, it can only be called in protected mode.
This routine can not be called in interrupt time.

ENTRY (AX:BX) = 32-bit physical address
EXIT carry set if error, clear otherwise
USES All registers preserved

FS and GS must be preserved (DevHelp requirement)

dh_AllocateGDTSelector

dh_AllocateGDTSelector - Allocate GDT selectors and linear address space

AllocateGDTSelector - Kernel interface for same

dh_AllocateGDTSelector is used to allocate
GDT selectors on behalf of a device driver.
This routine can not be called interrupt time.

ENTRY (ES:DI) = pointer to array to store selectors
(CX) = number of GDT selectors requested
EXIT Carry Flag clear on success
Carry Flag set on failure
(AX) = ERROR_INVALID_ADDRESS
ERROR_ZERO_SELECTORS_REQUESTED
ERROR_NOT_ENOUGH_SELECTORS_AVA

USES Flags
also AX on failure

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

Call DevhlpAllocGDTSel
return

dh_FreeGDTSelector

dh_FreeGDTSelector - Free GDT selectors

FreeGDTSelector - kernel interface for same

dh_FreeGDTSelector frees the GDT selectors allocated via
dh_AllocateGDTSelector.
This routine can not be called at interrupt time.

ENTRY (AX) = GDT selector to be freed
EXIT Carry clear if success

```

        Carry Flag set on failure
        (AX) = ERROR_INVALID_PARAMETER
              ERROR_ACCESS_DENIED
USES    EAX, Flags (Also DS and ES if they are equal to selector being
        freed.

PseudoCode:

Call DevhlpFreeGDTSel
return

```

dh_PhysToGDTSelector

dh_PhysToGDTSelector - store physical address in GDT selector

PhysToGDTSelector - kernel interface for same

Converts a physical memory address to a designated GDT selector. The indicated segment must be a fixed block of memory or locked (either long or short term) before issuing this request.

This routine can be called in interrupt time. It must not block, it must not get any faults.

Once this call has been issued for a particular selector, that addressability will remain valid until the device driver changes its content via a subsequent dh_PhysToGDTSel request referencing the same GDT selector. This call can be made in protected mode only.

```

ENTRY    (AX:BX) = Address - 32 bit physical address
        (CX)    = Size - the 32 bit segment size, not the limit
                  Converted to a limit before editing descriptors
                  A value of zero indicates 65536 bytes
        (SI)    = Selector - for the descriptor to be setup
EXIT     Carry Flag clear on success
        Carry Flag set on failure
        (AX) = ERROR_INVALID_ADDRESS
              ERROR_INVALID_PARAMETER
              ERROR_INVALID_SELECTOR
USES     EAX on failure
        Flags

        FS and GS must be preserved (DevHelp requirement)

```

PseudoCode:

```

Call PhysToGDTSel(Ring 0 data)
Return

```

dh_PhysToGDTSel

dh_PhysToGDTSel - store physical address and access in GDT selector

Converts a physical memory address to a designated GDT selector, setting the access byte of the descriptor to desired privilege value. The indicated segment must be a fixed block of memory or locked (either long or short term) before issuing this request.

This routine can be called in interrupt time. It must not block, it must not get any faults.

Once this call has been issued for a particular selector, that addressability will remain valid until the device driver changes its content via a subsequent `dh_PhysToGDTSel` request referencing the same GDT selector. This call can be made in protected mode only.

```
ENTRY    (EAX)    = Address - 32 bit physical address
        (ECX)    = Size - the 32 bit segment size, not the limit
                Converted to a limit before editing descriptors
                Must be <= 64Kb
        (DH)     = Descriptor privilege
                0: Make segment readable code (286 format)
                1: Make segment writable data (286 format)
                3: Make segment readable IOPL code (286 format)
                4: Make segment writable IOPL data (286 format)
                5: Make segment readable Ring 0 code (286 format)
                6: Make segment writable. Ring 0 data (286 format)
                128: Add to any of the above to get access privileges
                    in the 386 chip format (32 bit addressing)
        (SI)     = Selector - for the descriptor to be setup
EXIT     Carry Flag clear on success
        (SI)     = Selector with the right RPL bits
        Carry Flag set on failure
        (AX)     = ERROR_INVALID_ADDRESS
                ERROR_INVALID_SELECTOR
                ERROR_INVALID_PARAMETER
USES     EAX, ESI, Flags

        FS and GS must be preserved (DevHelp requirement)

PseudoCode

Call PhysToGDTSel
return
```

dh_GetDescInfo

dh_GetDescInfo - Return descriptor information

`dh_GetDescInfo` returns descriptor information for the given selector.

```
ENTRY    (AX) = selector
EXIT     Carry clear if descriptor valid
        (AH) = attribute byte(only Big, Granularity flags)
                ( = parameter count if call gate)
        (AL) = access byte
        (ECX) = 32 bit address stored in descriptor
                (CX = selector if Call Gate)
        (EDX) = 32 bit limit of descriptor
                ( = 32 bit offset if call gate)
        Carry set if descriptor invalid
USES     EAX, ECX, EDX, and DS, ES if not set to FLAT

        FS and GS must be preserved (DevHelp requirement)
```

dh_PhysToVirt

dh_PhysToVirt - Convert physical address to virtual for Device Drivers

```

*** PhysToVirtDM - convert physical address to virtual for kernel.
*** PhysToVirt - Convert physical address to virtual for old v1.0 code.

PhysToVirt converts a physical address to virtual by mapping one of
the reserved GDT selectors. This routine can be called at
interrupt time. It must not block, must not take a page fault.

ENTRY      (AX:BX) = Physical address
           (CX) = Length
           (DH) = 0 put the result in (DS:SI)
           1 put the result in (ES:DI)
EXIT       CF = 0, ZF = 0 (Will never cause a mode switch)
           If input DH = 0, converted address in DS:SI
           If input DH = 1, converted address in ES:DI
USES      DS, ESI if input DH = 0
           ES, EDI if input DH = 1

           FS and GS must be preserved (DevHelp requirement)

PseudoCode:

Save all registers
Get Current Interrupt Level
Convert Cur. Int. Level to index into PhysToVirt selector table
; Adjust physical address for page alignment
Set pageoffset = (physaddr : PAGEMASK)
Set lastaddr = (physaddr + size + pagesize - 1)
Set mapphysaddr = physaddr : not PAGEMASK
Set mapsize = lastaddr - mapphysaddr
If (output goes to ES), increment index into selector pool
Set sel= Selector value in selector pool
Get GDT Address
Read linear address of GDT.sel descriptor, mask off page offset bits
Call PGMapVirt ; map physical pages to linear
; Adjust GDT selector's base linear address
Set GDT.sel's d_loadaddr += pageoffset
If (output in DS)
    Set DS=sel, ESI = 0
Else
    Set ES=sel, EDI=0
Endif
Clear Carry, zero ; We don't do mode switches any more
restore all registers
return

```

dh_UnPhysToVirt

dh_UnPhysToVirt - UnPhysToVirt for Device drivers.

```

*** UnPhysToVirtDM - UnPhysToVirtDM for kernel.

This routine, companion of PhysToVirt is actually unnecessary
because we never do mode switching. (VDMs run in protected mode)
It is supplied for compatibility reasons.

ENTRY      None
EXIT       Carry, zero flag clear

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

return (carry clear, Zero flag clear)

```

dh_VirtToLin

dh_VirtToLin - convert virtual address to linear

dh_VirtToLin converts given selector:offset pair to flat linear address. It returns error if selector is not valid or if offset is beyond selector limit.

```
ENTRY    (AX)    = selector
         (ESI)   = offset
EXIT     C clear
         (EAX)   = linear address
         C set if failed
         (EAX)   = Error code
USES     EAX, EFlags

         FS and GS must be preserved (DevHelp requirement)

PseudoCode:

Call VirtToLin
```

dh_LinToGDTSelector

dh_LinToGDTSelector - convert linear address to virtual.

dh_LinToGDTSelector converts linear address to virtual (selector:offset) address by mapping the passed selector's ptes to the physical memory referred to by the passed linear address. Once this procedure is called for a specific selector, the mapping will remain valid until the selector is mapped to another address range via LinToGDTSelector, PhysToGDTSelector, or PageListToGDTSelector.

```
ENTRY    (EBX) = Linear address (must be on a page boundary)
         (ECX) = Length ( <= 64K)
         (AX) = Selector returned by dh_AllocGDTSelector
EXIT     C clear if conversion performed.
         C set if not
         (EAX) = Error code
USES     EAX on error, EFlags

         FS and GS must be preserved (DevHelp requirement)

PseudoCode:

Call LinToGDTSel
```

dh_PageListToGDTSelector

dh_PageListToGDTSelector - map selector to physical pages

Maps physical addresses described in an array of pagelist_s structures to a GDT selector setting the access byte of the descriptor to the desired privilege value. The physical pages described in the pagelist array must be allocated as resident or locked (either long or short term) before issuing this request.

This routine can be called in interrupt time. It must not

block, it must not get any faults.

Once this call has been issued for a particular selector, that addressability will remain valid until the device driver changes its content via a subsequent call to `dh_PageListToGDTSelector`.

FS and GS must be preserved (DevHelp requirement)

```
ENTRY    (AX)    = Selector to map
          (ECX)   = Sum of the pl_cb fields in the PageList array
                  Must be less than 64k
          (DH)    = Descriptor privilege
                  0: Make segment readable code (286 format)
                  1: Make segment writable data (286 format)
                  3: Make segment readable IOPL code (286 format)
                  4: Make segment writable IOPL data (286 format)
                  5: Make segment readable Ring 0 code (286 format)
                  6: Make segment writable. Ring 0 data (286 format)
                  128: Add to any of the above to get access privileges
                      in the 386 chip format (32 bit addressing)
          (EDI)   = pointer to array of pagelist_s structures to map
                  the selector to.
EXIT     Carry Flag clear on success
          (AX)    = Selector with the right RPL bits
          Carry Flag set on failure
          (EAX)   = error code
```

PseudoCode

PhysToGDTSel

PhysToGDTSel - store physical address and access in GDT selector

Converts a physical memory address to a designated GDT selector, setting the access byte of the descriptor to desired privilege value. The indicated segment must be a fixed block of memory or locked (either long or short term) before issuing this request.

This routine can be called in interrupt time. It must not block, it must not get any faults.

Once this call has been issued for a particular selector, that addressability will remain valid until the device driver changes its content via a subsequent `dh_PhysToGDTSel` request referencing the same GDT selector. This call can be made in protected mode only.

```
ENTRY    (EAX)    = Address - 32 bit physical address
          (ECX)   = Size - the 32 bit segment size, not the limit
                  Converted to a limit before editing descriptors
                  Must be <= 64Kb
          (DH)    = Descriptor privilege
                  0: Make segment readable code (286 format)
                  1: Make segment writable data (286 format)
                  3: Make segment readable IOPL code (286 format)
                  4: Make segment writable IOPL data (286 format)
                  5: Make segment readable Ring 0 code (286 format)
                  6: Make segment writable. Ring 0 data (286 format)
                  128: Add to any of the above to get access privileges
                      in the 386 chip format (32 bit addressing)
          (SI)    = Selector - for the descriptor to be setup
EXIT     Carry Flag clear on success
          (SI)    = Selector with the right RPL stamp
          Carry Flag set on failure
          (AX)    = ERROR_INVALID_ADDRESS
                  ERROR_INVALID_SELECTOR
                  ERROR_INVALID_PARAMETER
USES     EAX, ESI, Flags
```

PseudoCode

If (LDT selector), return `ERROR_INVALID_SELECTOR`

```

Check for invalid flags parameter
Read linear address from descriptor
If ( not (sel.d_attr :: D_UVIRT) )           ; must have UVIRT bit set
    return (ERROR_INVALID_SELECTOR)
Endif
Callfar VMGetOd
If (not D_UVIRT || owner != PTOGDTOWNER)
    return (ERROR_INVALID_SELECTOR)
Endif
If ( (physaddr + size -1) > 4Gb)
    return (ERROR_INVALID_ADDRESS)
Endif
; We have to adjust for the fact that physical address may not
; begin at a page boundary.
Set pageoffset = physaddr :: PAGEMASK        ; save offset into page
Set lastbyte = physaddr + size -1
Set mapsize = (lastbyte - physaddr + (pagesize -1)) :: not PAGEMASK
Set descriptor's loaddr += pageoffset        ; adjust descriptor base
Set descriptor to given access privilege
Set descriptor's limit to size-1
Read laddr from descriptor's address fields
Call PGMapVIRT(physaddr :: PAGEMASK, laddr :: not PAGEMASK,
    npages = mapsize/pagesize, flags)
If (error), InternalError
Return

```

UVirtToPhys

UVirtToPhys - convert a UVIRT address to physical address

When in protected mode, UVirtToPhys converts a selector-offset pair to a 32-bit physical address. Callable at interrupt time on GDT selectors only.

```

ENTRY    (AX:BX) = UVIRT virtual address

EXIT     'C' Clear if no error
          AX:BX = 32-bit physical address
          'C' Set if error

USES     EAX, EBX, DS, Flags

PseudoCode:

If (AX is an LDT selector)
    If (Interrupt time), return (carry )
    Convert sel:offset to linear address
Else
    ; GDT selector
    Get GDT Address
Endif
Convert selector to offset
Add sel offset to descriptor table's address
If (descriptor D_UVIRT bit not set)
    return (carry)
Endif
Read address fields in descriptor
Call dh_VMLinToPhys
Write physical address to AX:BX
return

```

VirtToLin

VirtToLin - convert virtual address to linear

VirtToLin converts given selector:offset pair to flat linear address. It returns error if selector is not valid or if offset is beyond selector limit.

```
ENTRY      (AX:ESI) = Virtual address (selector:offset)
EXIT       C clear
           (EAX) = linear address
           C set if failed
           (EAX) = Error code
USES       EAX, ECX, EDX, Flags

           FS and GS must be preserved (DevHelp requirement)

PseudoCode:

Load access byte
If (not segment), return error
Load limit
If ((expdown :: offset <= limit) || (not expdown :: offst > limit))
    return error
Call SELVirtToLin(offset, sel)
return (laddr)
```

LinToGDTSel

LinToGDTSel - convert linear address to virtual.

LinToGDTSel converts linear address to virtual (selector:offset) address by mapping the passed selector's ptes to the physical memory referred to by the passed linear address.

```
ENTRY      (EBX) = Linear address (must be on a page boundary)
           (ECX) = Length ( <= 64K)
           (AX) = Selector returned by dh_AllocGDTSelector or PhysToVirt
EXIT       C clear if conversion performed.
           C set if not
           (EAX) = Error code
USES       EAX, ECX, EDX, Flags

           FS and GS must be preserved (DevHelp requirement)
```

```
PseudoCode:

Call _GetDescInfo(sel)
If (not D_UVIRT), return (Carry)
For each page in selector, call PGPageToPhys and PGMapVirt to map to
    given linear range.
If (error), return (carry)
Set up descriptor's access byte (ring 0 data)
Set up descriptor's limit = size - 1
Turn off low 12 bits in descriptor's low address field
return
```

PhysToVirtInit

PhysToVirtInit - Initialize PhysToVirt variables.

Get the GDT selectors needed for PhysToVirt, also reserve 68Kb of linear address space and allocate corresponding page table entries. A 68Kb linear address space is allocated to each selector, in order to account for the fact that we can only map page boundary addresses to page tables and the physical address may not be on a page boundary. In that case we round down the physical address to nearest page boundary while mapping to page table and add the page offset into descriptor's base address field. The linear addresses are assigned to selectors permanently, and should never be erased or changed (except for page offset modifications).

```
ENTRY:  NONE
EXIT:   NONE
USES:   Flags
```

PseudoCode:

```
Set numsels= PTOV_NUMSELS          ; No of sels to be allocated
Set psel = offset of DOSGROUP:PTOVSelectors
Do (numsels) times
    Call VMAlloc (GDT space, UVIRT+RESIDENT, 68K, SELMAP, ring3 data,
                  PTOVIRTowner)
    If (error), InternalError
    Write Selector value to; psel
    Increment psel
return
```

PhysToVirtReinit

PhysToVirtReinit - Re-initialize PhysToVirt variables

This routine is called after system initialization to re-initialize any necessary PhysToVirt data. It sets the dpl to 0 in access bytes for all PhysToVirt descriptors, and forces any later descriptors to be dpl/rpl 0.

```
ENTRY:  NONE
EXIT:   NONE
USES:   All registers preserved          @@
```

PhysToUVirt

PhysToUVirt - convert physical to protected mode LDT virtual address.

PhysToUVirt converts a 32-bit address to a valid selector-offset pair.

Callable in user and kernel modes.

If the given physical address does not start on a page boundary, we will InternalError for now in order to straighten out all kernel callers, later on we will simply return carry.

```
ENTRY:  paddr = 32-bit physical address
        On flags=2 high word is LDT selector to free
        size   = size of segment
        flags   = 0: make segment readable code
                1: make segment writable data
                2: free selector
                3: make segment readable IOPL code
```

```

                4: make segment read/write IOPL data
                5: Tag selectors PVB, make segment writable data
(SI)           = 0 add/remove access from segment (Used only if dh=5)
owner         = used for ownership tracking
EXIT: Carry Flag clear on success
              (ES:BX) = valid virtual address
              Carry Flag set on error

USES:  EAX, EBX, EDX, ES, Flags

PseudoCode:

If (free request)
  If (not user space) return (carry)
  Get access rights (LoadAR)
  If (not D_SEG || Ring 0 || not D_W/D_X)
    return (Carry)
  Endif
  Call VMOBJInfo
  If (not UVIRT || owner != PTOUVIRTOWNER)
    return (Carry)
  Endif
  Call VMFreeMem(laddr, current context)
Else
  ; allocate
  If (size == 0), Set size = 64K
  If (physaddr + size - 1 > 4Gb)
    return (Carry)
  Endif
  If (physaddr is in the SGS control range), set VMAC_SGS flag
  Call VMAlloc(RESIDENT+UVIRT, SELMAP, privuser space, blockno=physaddr)
  If (error), return (Carry)
  If (VMAC_SGS flag is set)
    Call VMSGSControl
    If (error), Call VMFree, return error
  Call SetRPL (laddr to sel)
  Set UVIRT bit in descriptor
  Return selector:0 in ES:BX

```

146867 12/15/95 Change Team Change PhysToUVirt to panic only in the DEBUG version and to send an error message back in the retail version when a selector has not been freed.

TranslateRAS

TranslateRAS - convert offset of RAS info in globalinfoseg to linear address usable by 32 bit callers of DosQueryRASInfo

```

Callable in user modes.

ENTRY:  RASvirtual = offset of globalinfoseg information
EXIT:   (EAX) - contains the 32 bit linear address
        (EAX) - contains NULL value if error occurs

USES:   EAX, EBX, ECX, EDX, ESI, Flags

```

Imported Interfaces

The following procedures and data, or their equivalents are needed to complete Selector Manager implementation.

- Global Data:
- PTDA data

ptda_handle
ptda_ldthandle
ptda_ldtaddr
ptda_ldtpgmap

- VMM Component:

BMPFree
BMPGet
BMPInit
VMAllocMem
VMARSetTiled
VMAttach
VMClearSem
VMEachMap
VMFreeMem
VMGetContext
VMGetLaddr
VMGetMemInfo
VMGetPTDAAAddr
VMGetSem
VMHandleMap
VMIsAttached
VMLinToPhys
VMLock
VMMapAlias
VMGetObjectInfo
VMReserve
VMSearchRef
VMSelAlias
VMSetOwnerSel
VMUnlock
VMVirtFree

- Page Manager Component:

PGAlloc
PGAttach
PGFree
PGMapVirt
PGPageToPhys
PGSetAccess

PGSetType

- Program Loader Component:

LDRGetSegment

LDRFreeNonResNamSegs

LDRUseNonResNamSegs

- Tasking Component:

ProcBlock

ProcRun

TKSetSSBase

- Signals Component:

CheckSignal

- Other Components:

SemRequest

SemClear

WrtToScrnClr

Design Constraints

To be added.

Design Review

To be performed.

Selector component Implementation

This page is intentionally left blank.

Internal Interfaces

This page is intentionally left blank.

Miscellaneous

SELAllocKHB - Allocate selector mapped block from a kernel heap.
SELFreeKHB - Free Selector-mapped block from a Kernel Heap
SELFuBuff - Fetch user buffer via a 16:16 pointer
SELGetKHB - Get Kernel Heap Block Info for Selector-Mapped Heap Block
SELifsAlloc - Allocate heap/segment object for IFS
SELifsFree - Free an IFS segment/heap object
SELifsNoSwap - Make a swappable segment/heap object resident.
SELifsRealloc - Reallocate an ifs owned segment/heap object.
SELMoveKHB - Move given heap block into resident heap.
SELReallocKHB - Realloc Selector-mapped Kernel Heap Block
SELRemoveAccess - Remove user level access from Devhlp objects.
SELSuBuff - Set user buffer via a 16:16 pointer
SELValidateKHBAddress - Validate given heap address, return sel, owner
selAllocLDTSub - Allocate LDT selector(s)
selAllocPhys - Allocate Fixed Block of Physical Memory
selFreeLDTSub - Free LDT selector, return total descriptor size
selFreePhys - Free physical memory allocated via DevHelp AllocPhys
selGetSeg - Get access to a sharable segment.
selGiveSeg - Give access to a sharable segment to another task.
selInitInfoSegLDT - Initialize task's InfoSeg
selInitLDTSub - Allocate and initialize LDT
selLockDD - Lock a DD segment permanently at init time
selReallocSub - do reallocation on behalf of w_AllocSeg, ReallocSeg, or w_ReallocHuge
selValidateLDT - Validate region of LDT covering given descriptors

_selGetLDTAddr

_selGetLDTAddr - Get linear address of LDT

_selGetLDTAddr returns linear address of LDT given context handle.

ENTRY hptda - context handle (0 if current)
RETURN laddr - Linear address of LDT belonging to ptda
USES EAX, ECX, EDX, Flags

PseudoCode:

```
If(current context)
    Get [ptda_ldtaddr]          /* From stack */
Else
    Call VMGetPTDAAAddr
    get [ptda_ldtaddr]
Endif
#ifdef SELSTRICT
    Call VMGetObjectInfo
    If(he_owner != SELLDTOWNER) Panic
#endif
return linear address
```

_selFreeGDTSUB

_selFreeGDTSub - Free GDT selector, return committed size

_selFreeGDTSub frees the GDT selector and returns its existing descriptor size. It generates an Internal Error under any error condition.

```
ENTRY    sel - selector to be freed
RETURN   ulong - Existing descriptor size
          WARNING: Invalid if it is an expand down segment.
USES     EAX, ECX, EDX, EFlags
```

```
cbDesc = selFreeGDTSub(sel)
```

PseudoCode:

```
clear GDT descriptor access field
Read/calculate descriptor size
link onto free list
```

selCheckFreeList

selCheckFreeList - Check the GDT Free List to make sure selector

to be freed is not already free.

selCheckFreeList checks to make sure the selector to be freed is not already on the free list.

```
ENTRY      EDX - Selector offset to be freed.
            ECX - Highest valid selector value
            EBX - Linear address of GDT.
EXIT       NONE (Panic if selector already free)
USES       Flags
```

PseudoCode:

```
Walk the whole free list to see if any linked descriptor matches
the selector value passed.
```

selConvertToExpandUp

selConvertToExpandUp - Convert expand down base and limit to expand up.

Given values of limit and address from an expand down descriptor, selConvertToExpandUp returns the corresponding expand up values for address and size (NOT limit!)

```
ENTRY      (AH) = Attribute byte (Only G and B bits needed)
            (ESI) = 32 bit expand down limit (byte granular)
            (EDX) = Expand down base address

EXIT       (ESI) = 32 bit expand regular size
            (EDX) = Linear address of first byte of segment
```

```

USES    EDX, ESI, Flags

Pseudocode:

Ifdef SELSTRICT
    If (state of G bit != B bit), Panic
    If ( G bit == 0)
        If (limit > fffe), Panic
    Endif
Endif
Expandup address = expdown address + limit + 1
If (G bit == 0)
    Size = ffff - limit          ; negate, dec si
Else
    ; G bit == 1
    Size = ffffffff - limit      ; negate, dec esi
Endif
return

```

selConvertToExpandDown

selConvertToExpandDown - Convert expand up base and size to expand down.

Given the starting address, size and G bit, selConvertToExpandDown returns the expand down descriptor address and limit values.

```

ENTRY    (AH) = Attribute byte (Only G bit needed)
         (ECX) = 32 bit regular size (byte granular)
         (EDX) = Expand up base address

EXIT     (ECX) = 32 bit expand down limit (NOT size!)
         (EDX) = Expand down base address

USES     ECX, EDX, Flags

Pseudocode:

Ifdef SELSTRICT
    If (G bit == 0)
        If (size > ffff) Panic
    Endif
Endif
If (G bit == 0)
    Limit = ffff - size
Else
    ; G bit set
    Limit = ffffffff - size
Endif
ExpanddownAddress = ExpandupAddress - 1 - limit
Return

```

_selEachDesc

_selEachDesc - Worker function for VMEachMap.

_selEachDesc sets/updates the descriptor(s) to match the parameters pointed to by _SELSetDescInfo argument stack frame. If alias update is requested, only the SELMGR alias update is needed, because VMM aliases do not get descriptor updates after a realloc, commit or decommit. This procedure must remain in resident code, because it may be updating aliases in the GDT

```

ENTRY    (SS:ESP) = pdata _SELSetDescInfo argument stack frame
         (SS:ESP + 4) = Starting page address, ipg
         (SS:ESP + 8) = cpq if from VMEachMap,
                        GDT sel if called from SELSetDescInfo directly
         (SS:ESP + 12) = pod, pointer to object description structure
         (SS:ESP + 16) = pgoff if from VMEachMap,
                        -1 if called from SELSetDescInfo directly

EXIT     NONE
USES     EAX, ECX, EDX, Flags

```

```
_selEachDesc(pdata, ipg, cpq/sel, hptda, pgoff/-1)
```

PseudoCode:

```

Load ipg, hptda, pgoff into registers
Set stack frame to SetDescInfo stack frame
If (ipg < 512Meg)
    Call GetLDTAddr(hptda)
    Convert selector to offset into Descriptor Table
    If (ipg == sd_laddr >> PAGESHIFT)
        Call helper
    Else
        ; it's a cs alias limit upgrade
        if (sd_limit > 64Kb)
            set descriptor's limit = 0xffff
        else
            set descriptor's limit = sd_limit
        endif
    Endif
Else
    ; address in GDT area
    if (pgoff = -1)
        ; called directly from SELSetDescInfo?
        set sel = cpq/sel ; use given selector if so
        call helper
    else
        ; called from VMEachMap,
        ; this is a MemMapAlias
        set sel = SELConvertToSelector ; find selector
        call selGDTlimit
    endif
Endif
return

```

selGate

selGate - Set up given selector as gate descriptor (SD_GATE)

selGate sets the passed LDT or GDT descriptor as a gate.

```

ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
         (DS:EBX) = pointer to LDT or GDT descriptor
EXIT     Carry clear if operation complete
         Carry set otherwise

```

PseudoCode:

```

Ifdef SELSTRICT
    If (dwordcount > 31) Panic
Endif
Set eax register = selector << 16 | offset.low
Set [ebx] = register
Set up eax = offset.high << 16 | access << 8 | dwordcount
Set [ebx + 4] = eax
return

```

selSwitchLDTsub

selSwitchLDTsub - switch LDT selector for a VDM

selSwitchLDTsub switches the LDT for for the current VDM to the LDT specified in the TSD.

```
ENTRY    NONE
EXIT     NONE
USES     EAX
```

selBaseAll

selBaseAll - Set up base selector to cover all of the object

selBaseAll is called by selLDTAll and selLDTLimit to set up the first selector of the object to map all of the object instead of the first 64Kb. This rarely used feature is needed for Windows 3.0 support. The descriptor's access, attribute and address fields must already be set. This procedure only updates limit, extlimit and granularity field.
WARNING: This procedure is in swappable code.

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          First 5 parameters needed.
          (DS:EBX) = pointer to LDT descriptor
EXIT     NONE
USES     EAX, ECX, EFlags
```

selBaseAllAlias

selBaseAllAlias - Set up selectors of an 'base selector map all' alias

selBaseAllAlias is called by selEachDesc to update the selectors of the 'base sel. map all' type of alias after a realloc or SetMem. This rarely used feature is needed for Windows 3.0 support.
WARNING: This procedure is in swappable code.

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          (EBX)    = pointer to object descriptor
EXIT     NONE
USES     EAX, EBX, ECX, EDX, ESI, EDI, EFlags
```

selLDTAll

selLDTAll - Set raw descriptor information (SD_ALL)

selLDTall sets the passed LDT descriptor to the passed values.
 If the limit is larger than 64K, It sets up consecutive
 selectors with 64K in size, and the last one with the remainder
 of size. The descriptors must be allocated in advance.
 WARNING: This procedure is in swappable code.

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          First 5 parameters needed.
          (DS:EBX) = pointer to LDT descriptor
EXIT     Carry clear if operation complete
          Carry set otherwise
USES     EAX, EBX, ECX, EDX, Flags
```

PseudoCode:

```
Set eax = extaddr, attr(D_BIG only), access, hiaddr
#ifdef SELSTRICT
  If (requested attr : not (D_BIG+D_UVIRT)), Panic
  If (laddr.low), Panic ; must be on 64K boundary
#endif
  If (limit >= 64K)
    Set edx = laddr.low, limit
  Else
    Set edx = laddr.low, ffff ; d_loaddr and d_limit fields
  endif
  Set numdesc = limit.high
  Do (numdesc) times {
#ifdef SELSTRICT
    If ([ebx.d_access] :: D_VALIDMASK) Panic ; desc must be unused
#endif
    Set [ebx] = edx
    Set [ebx+4] = eax
    Set ebx = ebx + SEL_INCR
    Increment d_hiaddr in al
    If (carry), Increment d_extaddr in eax
  }
  If (limit.low)
    Set dx = limit.low
    Set [ebx] = edx
    Set [ebx+4] = eax
  endif
  return
```

selGDTall

selGDTall - Set raw descriptor information (SD_ALL)

selGDTall sets the passed GDT descriptor to the passed values.
 It assumes that expand down values are passed if the descriptor
 is expand down.

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          First 5 parameters are needed.
          (DS:EBX) = pointer to GDT descriptor
EXIT     Carry clear if operation complete
          Carry set otherwise
USES     EAX, ECX, EDX, Flags
```

PseudoCode:

```
Set eax = extaddr, attr, access, hiaddr
If (limit > 1Meg :: not (D_DATA :: D_EXPDN))
  turn on D_GRAN4K in attr
endif
If (D_GRAN4K)
  shift left limit by 12
endif
Set eax = eax : not D_EXTLIMIT
Turn on ext limit field in eax by high bits of limit
Set edx = loader, limit.low
```

```

Set [ebx] = edx
Set [ebx + 4] = eax
Return

```

selLDTlimit

selLDTlimit - Set LDT descriptor's limit field (SD_LIMIT)

selLDTlimit sets the passed descriptor to the passed values.
 If given newlimit is larger than 64K, it sets up consecutive
 selectors with 64k-1 limits and the last one with the remainder
 of the size.

WARNING: This procedure is in swappable code

```

ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
         (DS:EBX) = pointer to LDT descriptor
EXIT     Carry clear if operation complete
         Carry set otherwise
USES     EAX, EBX, ECX, EDX, EDI, Flags

```

PseudoCode:

```

Call selSDMatch          ; check base laddr, D_SEG,
If (error), return error
Set numdesc = newlimit.high
Do (numdesc) times
    Set ebx.d_limit = 64K-1
    Set ebx = ebx + SEL_INCR
If (newlimit.low)
    set [ebx.d_limit] = newlimit.low
    increment numdesc
    Set ebx = ebx + SEL_INCR
Endif
Set olddesc = oldlimit.high
If (oldlimit.low) Increment olddesc
If (olddesc > numdesc)
    Do (olddesc - numdesc) times
        Set [ebx] = 0
        Set [ebx + 4] = 0          ; zero out the escriptor
        Set ebx = ebx + SEL_INCR
    Else if (numdesc > olddesc)
        Set ebx = ebx - (numdesc - olddesc) *SEL_INCR ; first new desc
        Set highdw = [ebx + 4]
        Set lowdw  = [ebx]
    Ifdef SELSTRICT
        If (lowdw.high) Panic
    Endif
    Do (numdesc - olddesc) times
        Set ebx = ebx + SEL_INCR
        Increment highdw.lowbyte      ; adjust d_hiaddr
        If (carry)
            Set highdw = highdw + 1 << 24 ; adjust d_extaddr
        Endif
        Set [ebx + 2] = lowdw.high
        Set [ebx + 4] = highdw
    Ifdef SELSTRICT
        If ([ebx.d_attr] :: D_EXTLIMIT+D_GRAN4K) Panic
    Endif
    Endif
return

```

selGDTlimit

selGDTlimit - Set GDT descriptor's limit field (SD_LIMIT)

selGDTlimit sets the passed descriptor's limit to the passed value. It assumes that expand down values are passed if the descriptor is expand down.

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          (DS:EBX) = pointer to GDT or LDT descriptor
EXIT     Carry clear if operation complete
          Carry set otherwise
USES     EAX, Flags
```

PseudoCode:

```
Call selSDMatch
If (carry) return
If (not (D_DATA :: D_EXPDN))
    Set [ebx.d_attr] =: not (D_GRAN4K)
    If (newlimit > 1 Meg)
        Set [ebx.d_attr] =| D_GRAN4K
    Endif
Endif
If (D_GRAN4K)
    Shift left limit by 12
Endif
Set [ebx.d_limit] = limit.low
Set [ebx.d_attr] =: not D_EXTLIMIT
Set [ebx.d_attr] =| limit.high
return
```

selSDMatch

selSDMatch - determine whether the descriptor matches the request.

Algorithm:

- Check if segment descriptor
- Check if address fields match old address
- Return carry if any one doesn't match

```
ENTRY    (SS:EBP) = pointer to SetDescInfo ArgVar stack frame
          (DS:EBX) = pointer to GDT or LDT descriptor
EXIT     Carry flag clear if descriptor matches
          Carry flag set if not
USES     EAX, flags
```

PseudoCode:

```
If (not ([ebx.d_access] :: D_SEG))
    return (carry)
Endif
Set base.low = [ebx.d_loadaddr]
Set base.high.lowbyte = [ebx.d_hiaddr]
Set base.high.highbyte = [ebx.d_extaddr]
If (base != laddr) return (Carry)
#ifdef SELSTRICT
    If (not([ebx.d_access] :: D_PRES)) Panic
Endif
return
```

selMapHigh

selMapHigh - Map given selectors into high linear addresses

selMapHigh allocates UVIRT page tables, maps them to descriptor's physical address, and maps the given selector to the returned linear address.

```
ENTRY      AX - Selector
           EDX - User access flag
                0 Turn User access bit off
                PG_U Turn user bit on
```

```
RETURN     NONE
```

```
USES       EAX, ECX, EDX, Flags
```

PseudoCode:

```
Read size from descriptor, round up to page boundary
Call VMAlloc(UVIRT+SYSTEM+PAGEALIGN, size, SELUVIRTOWNER)
Read address from descriptor
Call PGMapPhys          ; map pte's to physical addresses
Store linear address in descriptor
```

ArenaSelToSeg

ArenaSelToSeg - Convert selector to physical paragraph address

ArenaSelToSeg maps a selector to a physical address/16

```
ENTRY      (AX)      = protected mode selector to translate
           (ES:DI) = arenainfo array
EXIT       (AX)      = matching paragraph address
           (ESI)     = matching linear address
USES       EAX, ESI, Flags
```

PseudoCode:

```
Given first element of arena info array:
If (a_oflags : OBJALIAS16)
    If (a_sel == given selector)
        read 32 bit physical address
        convert it to paragraph address
        return paragraph address
    Endif
Endif
get next element in arena info array
go back to checking for OBJALIAS16 flag
```

GDTMapInit

GDTMapInit - map initially allocated OS2KRNL and OS2LDR segments

Walk the ArenaInfo array to find all segments that need selector mapping, and map the selectors given in the ArenaInfo array. GDTMapInit maps selected DOS, OS2LDR, mini-FSD, remote IPL data, and other segments as indicated by the arenainfo array built by OS2LDR.

For R0CS or R0DS segments, adjust the selector's base address

so that the offset values will correspond to same values as it would when they move to the high end of the linear arena and have 0 base addresses.

The memory map will be as follows:

+-----+ Top of High Memory	
Ripl data	
+-----+	
mFSD.FSD	
+-----+	
free	
+-----+	
invalid	optional invalid physical address range(s)
+-----+	
free	optional secondary free range(s)
+-----+	
OS2KRNL(hi)	high kernel memory objects
+-----+	bottom of High Memory (at 1Meg)
:	screen, ROMs, etc.
+-----+	Top of Low Memory (usually 639k or 640k)
OS2LDR	
+-----+	
unused	
+-----+	
OS2KRNL(lo)	low kernel memory objects
+-----+	2000h
DosHlps	
+-----+	1000h
unused	rest of page 0
+-----+	
reserved	ROM data area, RM IDT
+-----+	0

```
ENTRY    (DS)    = GDT
          (ES:DI) = ArenaInfo array
          SELInitRM local variable frame
EXIT     NONE
USES     EAX, EBX, ECX, EDX, SI, DI, Flags
```

GDTMapAlloc

GDTMapAlloc - allocate a GDT selector for GDTMapInit and SELInitRM.

GDTMapAlloc allocates specific selectors as requested by the caller, and moves a dynlink call gate selector if necessary.

```
ENTRY    (SI) = selector to allocate
          (DS) = GDT
          SELInitRM local variable frame
EXIT     NONE
USES     Flags
```

GDTAllocInit

GDTAllocInit - Initialize the GDT pool of free descriptors.

GDTAllocInit links up the free descriptors in the GDT pool, and remaining descriptors at the end of GDT.

```
ENTRY    (DS) = GDT
          (FS) = DOSINITDATA
EXIT     NONE
```

USES AX, BX, CX, DX, SI, Flags

PseudoCode:

Link up the free descriptors, put head of link list in GDTFreeList.

GDTAllocN

GDTAllocN - Allocate GDT selector(s).

GDTAllocN allocates selectors (and descriptor) from the GDT.

ENTRY (ES:DI) = pointer to array to store allocated selectors.
 (CX) = number of GDT selectors to be allocated.
EXIT NONE
USES AX, CX, DI, Flags

SegLockSub

SegLockSub - Lock given range

SegLockSub locks given memory range on behalf of SegLock

ENTRY (ESI) = Flags
 (EBX) = Laddr
 (EDI) = Size
EXIT (EAX) = Error code
 (ECX) = ptr to lock handle stored in SELLockhandles object
 (if EAX = NO_ERROR)
USES EAX, EBX, ECX, EDX, EFlags

PseudoCode

Call BMPGet2
If (no error)
 Call VMLockMem
 If (error)
 Call BMPFree
 Endif
Endif

SegUnlockSub

SegUnlockSub - Unlock a lock done by SegLock

SegUnlockSub is called by SegLock and SegUnlock to unlock a lock done by SegLock.

ENTRY (ESI) = ptr to BMP32 selLockHandle
EXIT (EAX) = Error code (NO_ERROR if no error)

```
USES    EAX, ECX, EDX, EFlags
```

```
PseudoCode:
```

```
Call VMUnlock  
If failed return error  
Call BMPFree  
return
```

selCheckAllocParms

selCheckAllocParms - Check the sanity of SegAlloc input parms

selCheckAllocParms checks the sanity of _SegAlloc input parameters

```
ENTRY    _SegAlloc input parms CX(ch and cl inverted), DX, SI, EDI  
EXIT     NONE, InternalError if error  
USES     EAX, EFlags
```

_SegAlloc

_SegAlloc - Allocate a 16 bit segment, handle and selector.

_SegAlloc allocates an LDT or GDT selector, a memory segment handle and possibly physical memory. This routine is called to set up the first access to a 16 bit segment that does not yet exist.

NOTE:SegAlloc only allocated GDT or private LDT, Ring 0 or 3 writable data segment. For any other memory, use VMAllocMem

```
ENTRY    (BX) = Selector - the selector to be used  
          Required only if SA_SPECIFIC set in SFlags,  
          caller cleans stack if necessary.  
          (EAX) = Size - the segment's size.  
          zero is an invalid size  
          (CL) = SFlags - selector allocation flags:  
          SA_GDT - GDT selector if clear  
          SA_LDT - LDT selector if set  
          SA_SPECIFIC - use specific selector at ((SS:SP)) (SA_LDT)  
          SA_ALLOCATED - selector at ((SS:SP)) already allocated  
          (CH) = AFlags - access flags  
          D_PRIV field set to D_DPL[0/3]  
          (DX) = TaskPTDA - handle to the PTDA  
          (SI) = HFlags - Handle/memory allocation flags  
          HF_FIXED set if fixed memory needed  
          HF_MOVABLE if movable memory needed  
          HF_SWAPPABLE set if swappable  
          HF_LOWMEM set if <1M memory needed (defaults to fixed)  
          HF_ZEROINIT set if memory must be zero initialized  
          HF_NONCONTIG non contiguous memory requested (only if fixed)  
          (DI) = OwnerID - the owner of the segment (Must be either a  
          system owner or context handle)  
EXIT      (AX) = Error code (NO_ERROR if success)  
          if (AX = NO_ERROR)  
            (CX) = Handle  
            (BX) = Selector  
USES      EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, Flags
```

_selffsSegRealloc

_selffsSegRealloc - C callable interface to ifs_SegRealloc

```
int _selffsSegRealloc(cb, sel)

ENTRY      cb      - new size
           sel      - selector
RETURN     EAX      = error code
```

SegCheck

SegCheck - Verify access to a system segment

SegCheck verifies realloc/free access to a system segment by IFS.

```
ENTRY      (BX) = selector
           (DX) = owner
           (SI) = handle
           (CX) = object flags
EXIT       Carry flag set if ifs access disallowed
           Carry flag clear if access allowed
USES       EFlags
```

selAllocSeg

selAllocSeg - allocate a data segment

selAllocSeg allocates a private or unnamed shared data segment in the 286 chip, 16 bit format.

```
ENTRY      (AX) = allocation flag bits:
           0: private segment
           1: GIVESEG segment
           2: GETSEG segment
           4: discardable segment
           8: shrinkable segment
           (BX) = size in bytes (0 = 64K)
           (EDX) = VMAC_PROT if allocate from protected region, 0 otherwise
EXIT       Carry clear
           (AX) = selector
           Carry set
           (AX) = error code
USES       EAX, EBX, ECX, EDX, EDI, ESI, EFlags
```

PseudoCode:

```
Turn off highest bit in input flags
If(inputflags and not WAS_FLAGSMASK)
    return ERROR_INVALID_PARAMETER
Endif
```

```

If (size == 0), set size = 64K
Call VMAllocMem(SEL_64K, Sel Size, [VMALLOC FLAGS],
    HOBPTDA, HOBOWNER, HOBMTTE, [SELMGR FLAGS], BLKNO, RETURN_BUFFER);
return(selector)

```

selReallocSeg

selReallocSeg - realloc a segment

This function is called to realloc 16 bit unshared segments and shared segments. It will fail on CS aliases and huge segments.

```

ENTRY    (AX) = selector
          (BX) = size in bytes (0 = 64K)
EXIT     Carry clear on success
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, EFlags

```

Pseudocode:

```

If (GDT elector), return (ERROR_ACCESS_DENIED)
Call SELGetOd(sel)
If (error), return (error code)
if(ob_flags == VMOB_DISC_SEG)          ; if a discardable segment
    if(KSEMRequestExclusive(Discardable Seg Sem4) == ERROR)
        return (error code)
    endif
endif
If (not segment desc.), return ERROR_INVALID_PARAMETERS
If (D_CODE || ring 0 data || D_BIG || D_UVIRT)
    rc = ERROR_ACCESS_DENIED
    Jump exit
If (an alias || sel addres != object address || maxsize > 64K)
    rc = ERROR_INVALID_PARAMETER
    Jump exit
Endif
If (long term locks)
    rc = ERROR_LOCKED
    Jump exit
Endif
If (huge)
    rc = ERROR_ACCESS_DENIED
    Jump exit
Endif
If (newsize == 0), set newsize = 64K
If (OB_SHARED :: not OB_SHRINKABLE :: newsize < cursize)
    rc = ERROR_ACCESS_DENIED
    Jump exit
Endif
if ( (rc = selReallocSub()) == NO_ERROR)
    if(ob_flags == VMOB_DISC_SEG)          ; if a discardable segment
        if(Rmp_Find(hob) == ERROR)      ; Find rmp record using hob as key
            Panic
        endif
        if(softlocks < 0xff)             ; if not locked forever
            softlocks += 1;
        endif
    endif
endif
exit: if(ob_flags == VMOB_DISC_SEG)          ; if a discardable segment
    Release semaphore
endif
Call VMClearSem
return( rc )

```

_SELAddDiscRec

_SELAddDiscRec - Add a discardable segments rmp record

`_SELAddDiscRec` adds a discardable segments rmp record which is used to contain the segments soft lock count.

```
ENTRY    NONE

EXIT     AX = 0, ES:DI = Address of discardable segment rmp record
         AX = Error Code

USES     EAX, ECX, EDX, Flags
```

_SELRemDiscRec

_SELRemDiscRec - remove a discardable segments rmp record

`_SELRemDiscRec` removes a discardable segments rmp record which is used to contain the segments soft lock count.

```
ENTRY    hob - Object Handle of discardable segment whose rmp
           record is to be deleted. If hob == HOBNULL then
           es:di is the address of the record to remove.

EXIT     NONE (Panic if error)
USES     EAX, ECX, EDX, Flags, ES

PseudoCode:

;Request exclusive ownership of the Discardable segments RMP sem4
while ( (rc = KSEMRequestExclusive(Discardable Seg Sem4))
       == ERROR_INTERRUPT)
if(rc != NO_ERROR)
    Panic
endif
if hob != HOBNULL
    if(Rmp_Find(hob) == ERROR)           ; Find rmp record using hob as key
        Panic
    endif
endif
Rmp_Remove(found or passed record) ; Remove found record from rmp seg
Release semaphore
```

selUnlockSeg

selUnlockSeg - Discard-unlock a segment

This function is called to unlock a segment to allow it to be discarded. We don't really have runtime allocated discardable segments any more, so this routine is a bit of an anomaly. We tried just validating the selector and then returning no error if it was ok,

but we found that some apps did their unlocking by going into a loop repeatedly unlocking a segment and waiting for an error code to be returned (in order to undo the effect of multiple locks). To keep these apps from hanging we've added an RMP segment to keep track of the soft lock count for each discardable segment, using the segments hob as the key.

```
ENTRY    (AX) = selector
EXIT     Carry clear on success
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, EFlags
```

Pseudo-code:

```
;Request exclusive ownership of the Discardable segments RMP sem4
If(KSEMRequestExclusive(Discardable Seg Sem4) == ERROR)
    Return ERROR_ACCESS_DENIED
Endif
Call selValidateLockSeg
If(Not a discardable segment)
    Goto Exit
Else
    If(Rmp_Find(hob) == ERROR) ; Find rmp record using hob as key
        Panic
    Endif
    If(SoftLocks == 0xff)      ; Segment locked forever
        Goto Exit
    Endif
    If(SoftLocks == 0)        ; Segment not locked
        Release semaphore
        Return ERROR_NOT_LOCKED
    Endif
    SoftLocks = SoftLocks + 1 ; Set softlock count
Endif
Exit:
Release semaphore
Return NO_ERROR
```

selLockSeg

selLockSeg - Discard-lock a segment.

This function is called to lock a segment to keep it from being discarded.
(See selUnlockSeg for notes on discardable segments)

```
ENTRY    (AX) = selector
EXIT     Carry clear on success
          Carry set on error
          (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, EFlags
```

Pseudo-code:

```
;Request exclusive ownership of the Discardable segments RMP sem4
If(KSEMRequestExclusive(Discardable Seg Sem4) == ERROR)
    Return ERROR_ACCESS_DENIED
Endif
Call selValidateLockSeg
If(Not a discardable segment)
    Goto Exit
Else
    If(Rmp_Find(hob) == ERROR) ; Find rmp record using hob as key
        Panic
    Endif
    If(SoftLocks == 0xff)      ; Segment locked forever
        Goto Exit
    Endif
    SoftLocks = SoftLocks + 1 ; Set softlock count
Endif
```

```

        SoftLocks = SoftLocks + 1 ; Set softlock count
    Endif
Exit:
    Release semaphore
    Return NO_ERROR

```

selValidateLockSeg

selValidateLockSeg - validate a segment for discard-lock

```

Validate a selector for discard lockability.
(See selUnlockSeg for notes on discardable segments)

For v1.1 and v1.2 compatibility, this procedure should return:
* NO_ERROR for local and global infosegs
* ERROR_ACCESS_DENIED for Ring 0 or 1 selectors.
* ERROR_DIRECT_ACCESS_HANDLE for R2 or 3 UVIRT selectors.
* ERROR_INVALID_PARAMETER for non base huge selectors.

ENTRY    (AX) = Selector
EXIT     Carry clear on success
         Zero Flag set means segment is not discardable
         Zero Flag clear means segment is discard lockable
         (AX) = HOB
         Carry set on error
         (AX) = error code
USES     EAX, EBX, ECX, EDX, EDI, ESI, EFlags

PseudoCode:

SELGetDescInfo(selector)
If (not ring 2 or 3 data)
    Return (ERROR_ACCESS_DENIED)
Endif
If (UVIRT selector)
    Return (ERROR_DIRECT_ACCESS_HANDLE)
Endif
If (Huge selector and not base)
    Return (ERROR_INVALID_PARAMETER)
Endif
If (Discardable segment)
    Clear CF and ZF
    Return (hob)
Else
    Clear CF and set ZF
    Return
Endif

```

selAllocHuge

selAllocHuge - Allocate a huge memory segment

selAllocHuge allocates a huge memory segment. We reserve for the maximum size, and then commit the initial requested size. Selectors are set up to show the requested size.

```

ENTRY    (AX) = NumSeg - number of 64k segments to allocate.
         (BX) = Size - size of the last segment to allocate.
         (CX) = MaxNumSeg - the maximum number of segments ever
               to be required by a w_ReallocHuge call (or 0).

```

```

        (DX) = allocation flag bits:
            0: private segment
            1: GIVESEG segment
            2: GETSEG segment
            4: discardable segment
            8: shrinkable segment
        (ESI) = VMAC_PROT if to alloc from protected region, 0 otherwise
EXIT    Carry Flag clear
        (AX) = Selector - the base selector of the newly
            allocated huge segment.
        Carry Flag set
        (AX) = error code
USES    EAX, EBX, ECX, EDX, ESI, EDI, EFlags

Pseudocode:

If (maxnumseg == 0), set maxnumseg=numseg + 1
    set maxnum = numseg + 1
Endif
Call VMAllocMem(HUGE, size, maxsize, SELMAP, RING3 DATA) ;reserve memory
return(error code or selector)

```

selReallocHuge

selReallocHuge - Reallocate a huge memory segment

```

selReallocHuge changes the size of a huge memory segment.

ENTRY    (AX) = NumSeg - number of 64k segments desired.
        (BX) = Size - the size of the last segment desired.
        (CX) = Selector - huge base selector.
EXIT    Carry Flag clear
        NONE
        Carry Flag set
        (AX) = error code
USES    EAX, EBX, ECX, EDX, ESI, EDI, EFlags

PseudoCode:

If (GDT selector), return (ERROR_ACCESS_DENIED)
Call SELGetOd(take semaphore)
if(ob_flags == VMOB_DISC_SEG) ; if a discardable segment
    if(KSEMRequestExclusive(Discardable Seg Sem4) == ERROR)
        return (error code)
    endif
endif
If (error, or not D_SEG, or selector's address is not base)
    rc = ERROR_INVALID_PARAMETER
    Jump exit
If (D_UVIRT || D_CODE || D_BIG || not RING2 or 3 DATA)
    rc = ERROR_ACCESS_DENIED
    Jump exit
Endif
If (locked)
    rc = ERROR_LOCKED
    Jump exit
Endif
If (OB_UVIRT || not OB_HUGE || an alias || not AR_SELMAP)
    rc = ERROR_ACCESS_DENIED
    Jump exit
Endif
If (HF_LALIAS)
    Call VMSELAlias(export)
    If (cs alias) InternalError
Endif
Call _SELGetSize
If (OB_SHARED :: not OB_SHRINKABLE)
    If (newsize < cursize)
        Call VMClearSem
        return (ERROR_ACCESS_DENIED)
    Endif

```

```

Endif
if (selReallocSub() == NO_ERROR)
    if (ob_flags == VMOB_DISC_SEG)      ; if a discardable segment
        if (Rmp_Find(hob) == ERROR)    ; Find rmp record using hob as key
            Panic
        endif
        if (softlocks < 0xff)          ; if not locked forever
            softlocks += 1;
        endif
    endif
endif
exit: if (ob_flags == VMOB_DISC_SEG)      ; if a discardable segment
        release semaphore
    endif
Call VMClearSem
return (error code from ReallocSub)

```

selCreateCSAlias

selCreateCSAlias - Create 16 bit code selector to

access a data segment

This function allocates a selector in the LDT, and sets it up so that it can be used as 16 bit code selector for the given data segment. The value of the 16 bit code selector is returned.

```

ENTRY    (AX) = Data Selector to be aliased.
EXIT     Carry clear on success
         (AX) = 16 bit code Selector
         Carry set on error
         (AX) = Error code
USES     EAX, EBX, ECX, EDX, ESI, EDI, EFlags

PseudoCode:

If (GDT selector), return ERROR_ACCESS_DENIED
Call SELGetOd(datasel, take semaphore)
If (error),
    return ERROR_INVALID_PARAMETER
Endif
If (not ring2 or 3 data)
    Call VMClearSem
    return (ERROR_ACCESS_DENIED)
Endif
If (shared, or discardable, or huge, or uvirt, or maxsize > 64Kb)
    Call VMClearSem
    return ERROR_ACCESS_DENIED
Endif
If (OB_LALIAS)          ; already aliased?
    Call VMSelAlias(export cs alias record address)
    If (no error)
        If ( (pla->cs_cref >> CS_REFSHIFT) < 1000 )
            Set pla->cs_cref += (1 << CS_REFSHIFT) ; Inc ref count
            Call VMClearSem
            return(cs selector and NO_ERROR)
        Else
            Call VMClearSem
            return(ERROR_NOT_ENOUGH_MEMORY)
        Endif
    Endif
Endif
; No cs aliases exist
Call VMMapAlias (laddr, 64K or datasel size, private LDT)
If (error)
    Call VMClearSem
    return (error code)
Endif
Call VMSelAlias

```

```

Set up cs_selcode, cs_seldata, cs_fs, and cs_cref fields
Set retcode = (NO_ERROR)
Write codesel value
Call VMClearSem
return(retcode)

```

_SELDiscSegInit

_SELDiscSegInit - Initialize the discardable segment

```

                                RMP segment and sem4.

_SELDiscSegInit calls Rmp_Alloc to initialize the
discardable segments RMP records segment and KSEMAlloc
to initialize the discardable segments semaphore.

ENTRY      NONE
EXIT       NONE
USES       EAX, ECX, EDX, Flags

```

GDTInvalidate

GDTInvalidate - Invalidate a GDT Selector

```

GDTInvalidate marks the descriptor associated with the passed
GDT Selector as unused. DS will be loaded with the null selector
if it will be invalidated by this procedure.

ENTRY      (AX) = Selector - GDT selector to be invalidated.
EXIT       (ES:BX) = Pointer to associated descriptor.
           (ES)    = GDT
USES       AX, BX, ES, Flags
           and DS if it matches the Selector parameter

Pseudo code:
    clear GDT descriptor access field
    link onto free list

```

DescInvalidate

DescInvalidate - Invalidate a descriptor

```

DescInvalidate marks the descriptor associated with the passed
Selector as unused. DS will be loaded with the null selector
if it will be invalidated by this procedure.

ENTRY      (AX) = Selector - GDT/LDT selector to be invalidated.
EXIT       NONE
USES       AX, BX, DX, SI, DS, ES, Flags

```

DevHlpAllocGDTSel

DevHlpAllocGDTSel - Allocate GDT selectors for a Device driver mapping

DevHlpAllocGDTSel is used to allocate permanent GDT selectors on behalf of a device driver. It will also allocate linear address space that these descriptors will map to. In order to account for the fact that we may have to map these to physical addresses that are not on a page boundary, we will allocate 68K of linear address space for each selector. This routine can not be called interrupt time.

ENTRY (ES:DI) = pointer to array to store selectors
(CX) = number of GDT selectors requested
(BX) = callers BP (used for ownership tracking)

EXIT Carry Flag clear on success
Carry Flag set on failure
(AX) = ERROR_INVALID_ADDRESS
ERROR_ZERO_SELECTORS_REQUESTED
ERROR_NOT_ENOUGH_SELECTORS_AVA

USES Flags
also AX on failure

FS and GS must be preserved (DevHelp requirement)

PseudoCode:

```
TASKTIME only
If (cx = 0), return ERROR_ZERO_SELECTORS_REQUESTED
If (ES is LDT selector)
    Convert ES value to laddr via laddr = sel >> 3 <<16
Else
    Get GDT Address
    Add ES offset to GDT base address
    Read linear address from descriptor
Endif
Add DI value to laddr
Call w_MemRegion(laddr, size=2*no of GDT selectors)
If (error),
    return ERROR_INVALID_ADDRESS ; user passed address is bad
Endif
Save numsels=no of selectors to allocate
Do cx times {
    Call VMAlloc (RESIDENT+UVIRT, SELMAP, system area, size = 68K
    PToGDTOwner)
    If (error)
        Do (numsels - cx) times {
            Decrement di by 2
            Read selector at es:di
            Convert GDT selector to laddr
            Call VMFreeMem
            return (ERROR_NOT_ENOUGH_SELECTORS_AVA)
        }
    Endif
    Store GDT selector value at es:di
    Increment di by 2
}
return
```

ClrInvSel

ClrInvSel - Macro to clear/restore segment registers

```
ClrInvSel clears or restores the value to a segment register
based on whether the initial given value of that register is
valid or not.
ENTRY      AX = 0
           sel    - Selector that is invalidated (!= CX)
           segreg - Segment register
           initval - Initial value of the segment register (!= CX)
EXIT       AX = 0, segreg restored to initial value or cleared
USES      segreg, CX
```

Implementation

To be added.

Implementation Review

To be performed.

Selector component Appendix (included for each review)

This page is intentionally left blank.

Glossary

To be added.

Size and Performance Considerations

To be added.

Implementation Estimates

	LOCs	Effort	ELOCs	MM
Redesign LDT	100	1.0	100	0.4
LDT allocated as a sparse object and pages validated as needed. Shared sels allocated from high end, priv.sels low				
SegValidate removal/SegDiscard/ NotPresentFaultHandler	20	1.0	20	0.08
32 bit descriptor limits	100	1.0	100	0.4
Get/SetDescInfo, SegGetInfo				
Add 32 bit address/offset support to all code.	100	1.0	100	0.4
Expand SegLockInterface to 32 bits	40	0.8	32	0.13
Virtual pointer allocation for get_segment, get_page	30	0.5	15	0.05
SM Initialization	20	1.0	20	0.08
SELEditDGROUP				
Changes to GDTInitRM				
TOTAL	410		387	1.54

Swap component

This page is intentionally left blank.

Swap component Architecture

This page is intentionally left blank.

Problem Description/Objectives

OS/2 will provide demand paged virtual memory. The Page Manager will control which pages are kept in physical memory and which will be swapped out. A Swap manager is required to control the transfer of pages between memory and secondary storage. Since there is one page per 4k of allocated virtual memory in the system, the page swapper must be able to manage large numbers, potentially tens of thousands, of pages without consuming excessive system resources. Historically page swapping has had a major effect on the performance of computer systems, so performance issues must be given full consideration during both design and implementation of the swapper. The OS/2 Swap manager must therefore meet the following requirements:

- Provide the Page Manager with interfaces to swap pages in and out of physical memory
- Use as little memory per page as possible
- Provide the Page Manager with information for overcommit calculations

- Be able to dynamically grow and shrink the swap file
- Be able to swap to non FAT-file-system devices

Solutions/Justification

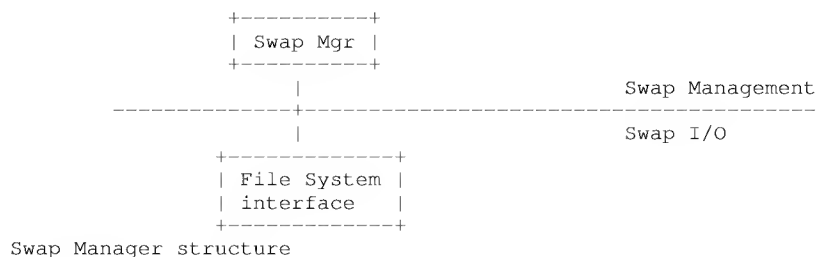
Although the functionality of the page swapper is largely determined by external factors, there are a number of possible approaches that can be used to implement the swapper.

1. Modify the OS/2 v1.0 segment swapper

This was done for a prototype 386 version of OS/2, with mixed success. Much existing code can be used with only minimal adaptation, but performance is very poor. The segment swapper was designed to manage hundreds, or at most thousands, of variable size objects instead of the thousands or tens of thousands of fixed size objects the page swapper must manage. As a result, it uses large per object data structures and has a high per swap operation overhead, including two context switches. Many simplifications, in both data structures and algorithms, can be applied by writing a swapper designed specifically for pages.

2. Write a purely page based swapper

The function of the swapper can be broken into two components: swap management and swap I/O. The Swap manager is responsible for managing space inside the swap file and translating swap in and out requests into calls to the swap I/O routines. The swap I/O routines do essentially nothing other than the actual read and write operations, and possibly resize the backing store. The advantage of this is that the two levels are largely independent making it easy to prototype several different versions of the swap I/O routines.



The swap management level would handle the swap-device-independent data structures, and provide the interface to the Page Manager. The major data structures would be an array of SFs, one for every page of swappable memory allocated in the system, and a bitmap to keep track of allocations within the swap file, with each bit mapping 4k of space in the swap file. For each SF the Swap manager records a reference count, a cross link to a PF, a DFs, which is an index into the bitmap (and hence the swap file), and some flags indicating the state of the page described the frame. The Swap manager would handle space allocation within the file and would also be responsible for keeping track of overcommit information.

The swap I/O level consists of a number of routines, each handling a simple I/O operation (read, write, change size, etc.). During system development it should be possible to prototype several different variations of the I/O routines. Several possible designs are described below.

a. Swap to a file through the file system

The simplest method of doing page swapping is to have the swapper open a file in a DOS disk partition and use the same kernel file I/O routines that the loader uses to do the necessary reads and writes. By doing file I/O using SFT-based calls rather than handle based calls, it is possible to swap in pages in the context of the faulting process, avoiding a context switch to a separate swap thread. Since the swapper would be competing with user applications for disk space, the swap file would be grown and shrunk as needed.

This method has the advantages of being relatively easy to implement and upwardly compatible with the 1.0 scheme. Since the pages are a uniform size there would be no fragmentation in the file, and it would be possible to shrink the swap file after a period of high transient memory usage had passed, a major usability improvement over 286 DOS-boxes. Performance would be much better than that provided by the segment swapper, although still less than optimal.

b. Swap directly to a separate partition

To avoid the (external) fragmentation problems and overhead inherent in reading and writing through the file system, all paging could be done to a separate partition. To avoid accidentally damaging existing file systems, a new partition type (usable only for paging) would have to be defined. This method would eliminate the need to use the file system and would allow for substantial performance improvements. The swapper would communicate directly with the disk device driver to do absolute sector reads and writes. This would eliminate most of the overhead associated with using the file system, including saving and restoring TCB variables, searching the buffer cache, and reading the FAT. It may also be possible to make many of the kernel file system data structures swappable, since they would not be needed by the swapper. While there would be some knowledge of disk layout embedded in the swapper, the hardware dependent code is entirely contained in the device driver, so that new device types would be supported automatically.

In order to swap via this method, users would have to repartition their hard disks to provide a paging partition. Also, the swapper would have to be prepared to deal with hard errors directly, instead of depending on the file system. The requirement that a disk be repartitioned, and hence all data on it lost, in order to change the amount of swap space available may prove to be a fatal usability problem, despite the anticipated performance improvement.

c. Swap to the file system using a streamlined file system interface

Since the swapper is part of the kernel and hence a "trusted" component, it would be possible to produce a streamlined file system interface that would do little if any argument validation. Although the Swap manager routines in this case would be nearly identical to those described in the file system option described above, performance could be expected to be better, since there would be no more overhead than would be incurred in swapping directly to a partition. The design of the streamlined file system interface is discussed elsewhere.

Writing a new page based swapper is the superior solution. The nature of page swapping is sufficiently different than segment swapping to make converting the old segment swapper an unsatisfactory alternative. The division between swap management and swap I/O is a natural one and adds no complexity to the design. Since the I/O routines do not deal with the higher level data structures, other I/O mechanisms can be prototyped easily.

Initially the system will be written to swap through the file system using standard kernel file I/O routines. Other systems can be prototyped, and when and if a streamlined file system interface becomes available, the swapper I/O routines can be converted to call the new entry points.

Error Handling

Error handling in the swapper is a somewhat delicate issue. When hard errors start occurring on the backing store it may be impossible to activate the hard error daemon, and swappable parts of the kernel may be lost forever. This makes it impossible to guarantee that the system will be able to continue running for any length of time, if at all. The goal in error handling, then, is to return any error back to the user, and keep the system from collapsing long enough for things to be shut down gracefully. Accordingly, hard error popups will be suppressed on swap operations, and any errors will be propagated back up to higher level routines and recorded via any kernel error logging mechanism that might be implemented. Swap operations will be retried, and failed only if both attempts fail. If a process has a swap operation fail, the DF where the error occurred will be marked as broken and, once freed, will never be reallocated. The overcommit code must be prepared to deal with the useful size of the swap file shrinking unexpectedly in this manner.

Architectural Review

To be performed.

Swap component Design

This page is intentionally left blank.

Design Overview

Since the Swap manager component is split into several distinct pieces, this overview will be broken down in a similar manner. The relationship between the various parts of the swapper has already been discussed in the component architecture section.

Swap Manager exports

The following variables are exported from the Swap manager to the rest of the kernel. In all cases, reading the value of the variable is allowed, but altering it is not, as this will adversely affect swapper behavior.

`byte TCB_fSwapping (TCB variable)`

This TCB variable is set to indicate that this thread is currently involved in a swap operation.

smHobheap
smHardErrPID
SMSwapperSem
SMcDFInuse - number of allocated DFs.
SMcInMemFile
SMpInMemFile
SMSwapSpace - current amount of swap space (in pages)
SMPageSemHeld - page semaphore owned by compaction daemon
SMCompactSlot - slot number of compaction daemon

The following functions can be called from anywhere in the kernel, subject to the restrictions listed in the header for each function.

SMSwapOut - Swap Managers group swap out entry point
SMSwapIn - swap a page in from disk to memory
SMSetSwapSpace - request a change in swap file size
SMQuerySwapSpace - estimate maximum possible swap space
SMReinit - finish up the ugly work of initializing the swapper

Several conversion functions are needed, e.g. to translate between byte counts and page counts. Since these conversions are relatively simple and will be done in-line in the final code, they are presented below in the form of macros.

RoundUpPages - round page count up to swap file granularity
SwapIdToByte - swap id to file offset conversion
ByteToSwapId - file offset to swap id conversion
PagesToBytes - page count to byte count conversion

The following data items are local to the Swap manager.

smpHeap
smMinSwapFile
smcGrowFails
smbmDF
smFileSize
smSizeHistory - history value used for tracking swap file size
smDFMac - highest allocated DF number
smcBrokenDF

SMMinFree
smPendingShrink
smInitialized - has swapper been initialized?
smHeap - handle for the swapper heap
smToe - used for timed swap file shrinking

The following functions are internal Swap manager workers, and are not externally visible.

smParanoid - enable super-strict checking
smSetSwapFlags - sets TCB swapper variable
smClearSwapFlags - clears TCB swapper variable
smAllocDF - obtain a disk frame
SMFreeDF - free a disk frame
smGRPAllocDF - allocate a group of Disk Frames
smIncreaseSpace - handle requests for increases in swap space
smDecreaseSpace
SMTomHook - shrink hook called by timer
smPerformShrink - perform a pending shrink, if possible

The following are Swap manager I/O routines, local in scope to the Swap manager, but implementing a different level of functionality.

data local to swapper SFT routines
smSFTmap - swapper SFT allocation bitmap
smSwapDisk - drive number of the drive containing the swap file
smPCarr - array of SMPageCmdList pointers
smSIOSwapIn - swap a page in
smSIOResize - resize the swap file
smSIOSpaceQuery - determine free space on the swap volume
smGrpSwpErrHandler - error handler for group swapping
smRetrySwapOut - single swapout after error on group swap out
smSetCmdl - set header and first cmd fields for a SMPageCmdList

*****LP_smAllocSFT - allocate a SFT from the swappers pool**

This very simple routine examines a tiny bitmap to find the first unused swapper SFT, and returns the index of the SFT. If there are no free SFTs, we block and try again later. If this is the case, then all the SFTs are in use by other threads doing swapping. We know that none of these threads will need another SFT to complete, so there is no danger of deadlock.

```

- find first free SFT

- update SFT bitmap

- if no free SFTs, block and loop back to try again

ENTRY   pIndex = addr of location to store SFT index
EXIT    (EAX) = error code
        NO_ERROR
        ERROR_INTERRUPT
        [pIndex] = index of allocated SFT in swapper SFT table

WARNINGS      can block
  
```

```

top:
    eax = index of first set bit in smSFTmap = index of first free SFT
    if no bits were set, goto wait
    store SFT index through parameter pointer
    clear the bit
    return NO_ERROR

wait:
    ProcBlock(interruptible, &smSFTmap)
    if (normal wakeup)
        goto top
    else
        return(ERROR_INTERRUPT)
  
```

*****LP_smHardError**

- smHardError cause a harderror to be displayed if the swap file cannot be extended (out of disk space).

```
ENTRY    NONE

EXIT     NONE

USES     EAX, ECX, EDX, EFlags
```

***LP_SMPrintFaultMsg

- SMFatalFault cause a system fatal fault and a message to be displayed when there is no more disk space left for swap file allocation and the entire swapfile is in use.

```
ENTRY    NONE

EXIT     Does not return

USES     EAX, ECX, EDX, EFlags
```

***LP_smFreeSFT - return an SFT to the swapper's pool

This routine returns an SFT to the swapper's pool, and runs anyone blocked waiting for an SFT.

- wack bit in SFT bitmap
- restart anyone blocked waiting for a swapper SFT

```
ENTRY    Index = index of SFT to return to pool
EXIT     none

set bit in smSFTmap
ProcRun (&smSFTmap)
```

Initialization

During system initialization the swapper will examine the value of the "swappath=" line in the config.sys file. If swappath is set equal to a directory specifier (e.g., "c:\", "c:\foobar") then the file SWAPPER.DAT will be created in the indicated directory and all swap I/O will be done through SFT-based calls to the file system. This will allow swapping to be done to an Installable File System transparently, with no special case support in the swapper.

The Swap manager must behave in a sensible manner when confronted with an incorrect or invalid swappath= entry in config.sys. There are several errors that are likely to be made, and a "Do What I Mean" strategy will be applied in handling them. If the user specifies a drive rather than a path (e.g., "swappath=d:"), this will be interpreted as meaning the root directory of that drive. If the the swap file cannot be opened in the requested directory, the swap file will be placed in the directory \os2\system of the boot device, and, failing that, in the root of the boot device. If none of these attempts succeed then swapping will be disabled. During the error handling process, informative error messages will be printed to the console.

Since we may wish to prototype swap I/O routines that take different kinds of addresses (virtual, physical, and potentially linear), during initialization the Swap manager will call the Page Manager to inform it which kind of addresses should be passed to SMSwapIn/SMSwapOut calls. After this point the Swap manager can ignore the address type, since it simply passes addresses on to the lower level routines without interpretation. Hopefully the "correct" address type will be agreed upon well in advance of shipping, and the page manager can be modified to pass that address type without having to check at every call (or without having to set up an alias, should physical addresses turn out to be

superior).

Swap manager

One of the Swap manager's major functions is to manage space usage in the backing store. It does this by maintaining the SF array, one cell for each page of swappable memory (see the figure), with the index of the cell (or, equivalently, its address) serving as an identifier, called the "swap id". Free SFs will be placed on a linked list, so that allocations can be performed in constant time by simply removing the first element from the list. The mapping of swap id to actual location on the backing store is handled via the DF field in the SF. While a given SF may not have any space in the swap file currently allocated to it, the swap file will be kept large enough that all swap out requests can be met. Updates to the SF array must be made atomically, to prevent multiple threads from making conflicting modifications. Since all SF updates will take place in kernel mode, atomicity will be guaranteed by simply not blocking until the update is complete. If it becomes necessary to have the routines block or yield (for latency considerations or other reasons) a semaphore will be used to serialize access to the SFs. Serialization for resizes of the entire SF array is provided by the Page Manager's overcommit semaphore.

Entry points will be provided for the Page Manager to allocate and free SFs, and by so doing setting up a permanent association between an SF and a given page. Allocation of space in the swap file will occur automatically in the SMSwapOut call. If no free DFs are available (as indicated in the bitmap), a DF will be stolen from some "idle" SF. A SF is said to be idle when the page has a representation both on disk and in memory, i.e. when it is linked to both a PF and a DF. Unlike the old segment swapper, swap space will not be automatically deallocated at during SwapIn calls. Instead, the SF will be placed on an "idle list". This will allow us to reclaim the swap space if needed, but will preserve the link to the swap image so that if the page is swapped out before it is dirtied again we can merely discard the memory image without having to write it to disk.

The Swap manager will be responsible for setting and clearing the TCB_fSwapping TCB variable around calls to the I/O routines. The need for and function of the TCB_fSwapping variable will be explained in greater detail in a later section, but basically it serves to indicate that a given thread is swapping, and is generally analogous to the SwapperProc PTDA variable used in OS/2 v1.0.

File System Swapper I/O routines

The most efficient way to use the file system from within the kernel is by calling routines that work at the SFT level. Since SFTs are unique within the system, SFT-based I/O may be done in the context of arbitrary processes. For the purposes of the swapper, this means that we will not have to switch to a specific 'swapper process' in order to perform swap I/O as was done in 286 DOS-boxes. During initialization, a number of SFTs will be allocated and all opened to refer to the swap file. These SFTs will be used exclusively for swapping, and therefore will never need to be closed or re-opened, since they refer to the same file for the entire time the system is running. The only operations that will need to be performed after initialization are seek, read, write and resize.

For SMSwapIn and SMSwapOut operations, the DF number will be multiplied by the size of a page (4k bytes, effectively a left shift by 12 bits) to determine the offset of the DF in the swap file. A file seek will be done to this position and then the read or write operation performed. I/O will always be done in page-sized and aligned units, so the transfer count will be known to be a multiple of 4k. The file system requires the buffer it reads or writes to be fixed in contiguous memory. Fortunately, individual pages are contiguous by definition, so this does not present a problem. The file system interfaces used currently take virtual 16:16 addresses, so during initialization the Swap manager will inform the Page Manager to provide addresses in this form.

Error Handling

When an error occurs on a SwapIn operation, if possible an error message will be written to the console via WrtToScrn, or whatever emergency notification system succeeds it in later versions of OS/2. This is necessary because the hard error daemon may no longer be functional, and cannot be relied upon (we may have encountered the error swapping in part of the hard error daemon, and invoking it again would deadlock the system). The offending disk frame will be marked as broken and the count of broken frames will be incremented. An error will then be returned to the Page Manager so that appropriate actions can be taken.

An error occurring on a SMSwapOut operation is handled in much the same way. The error will be logged to the screen, the frame marked as broken, and the count of broken swap frames incremented. The DF will be marked as broken and abandoned, never to be used again. Since

the Page Manager cares only about the SF, it need never know that an error occurred.

If FreeSF is passed a SF that is linked to a broken DF, the DF will not be marked as free, but instead will simply be ignored. This will prevent the frame from being accidentally reused. Similarly, if the Swap manager idles a SF with a broken DF, the DF will be silently freed. When a disk frame goes bad the variable SMSwapSpace will be decremented, to indicate the reduced amount of swap space available.

Typical Execution Paths

Now that the components of the Swap manager have been described separately, it's time to show how they work together to handle swap requests. The simplest case is that of a swap-in operation. The Page Manager calls SMSwapIn, a part of the Swap manager, with a swap id and an address for the page to be loaded at. Swap-ins don't affect allocation or overcommit, so the Swap manager has very little work to do. It simply calls the swap in I/O routine, passing in the memory address and the DF associated with the SF it was passed. The swap-in routine translates the DF into a location in the backing store, performs the read, and returns. Assuming the read was successful, the Swap manager then adds the SF to the idle list and propagates any error code back to the Page Manager.

A swap-out operation is little more complex. The Page Manager calls the Swap manager at SMSwapOut with a request to swap out a page, specifying both the address of the page in memory and the SF representing it. The Swap manager If the SF does not already have a DF associated with it then the Swap manager allocates one. The I/O request is then passed on to the lower level routine, which performs the write and returns an error code. If the page was swapped out successfully, the SF is marked as holding a clean copy of the page.

Since swap-in and swap-out operations do not automatically allocate and deallocate SFs, the Page Manager must explicitly call SMAAllocSF to allocate a SF before a swap-out is attempted, and call FreeSF to dispose of the SF when it is no longer needed. Typically when a call is made to SMAAllocSF the Swap manager will simply take the first SF off of the free list and return its id. Overcommit calculations will prevent this call from ever returning an out of memory error.

The third major entry point in the Swap manager is the SMSetSwapSpace function. When the Page Manager wants to change the total amount of swap space available, it calls SMSetSwapSpace indicating the new amount of swap space it requires. If this number represents an increase, and is larger than the current value of SwapSpace, the swap file will be grown. Generally the swap file will be grown in fairly large quanta (such as 512k), so that it will not be necessary to grow again for a while. SwapSpace will be updated to reflect the new size, and SMSetSwapSpace will return. If the SMSetSwapSpace represents a decrease in required swap space then there are two possibilities. If the swap space requested is still "close" to the value of SwapSpace, then no action will be taken and RequestSwapSpace will return success. If the swap space requested is substantially less than the current SwapSpace, then an attempt may be made to shrink the backing store. A new value for SwapSpace will be chosen that is large enough to permit some growth in the requested swap space before the the backing store would have to be grown, in an attempt to keep SwapSpace from bouncing unnecessarily. Once the new value of SwapSpace has been chosen, the backing store may or may not be resized immediately. If any DFs are in use past the SwapSpace boundary, the Swap manager will have to wait until they have been freed before it can effect the actual resizing. In this case a "pending resize" flag will be set. This flag will be examined when freeing DFs, and if it is set and all frames past SwapSpace have been freed, the swap file will be shrunk and the flag reset.

The final major entry point is SMSetcSF, which the Page Manager calls to control the number of SFs available for allocation via SMAAllocSF. When an increased number of SFs is requested, the Swap manager will attempt to commit sufficient memory to handle the request. If the request is a decrease then unneeded SFs will be gather up into a separate list (the "shrink list") and decommitted at a convenient point.

Restrictions and Limitations

There are some situations in which other components want to know how much swap space can be made available, but does not want to expand the backing store to this size immediately. The QuerySwapSpace function will return an estimate of the maximum possible size of the backing store, but with no guarantee that that much space will actually be available when needed. The problem is that in a multitasking system the concept of "maximum size" is somewhat nebulous. Free space on the logical device containing the swap file will be queried and added to the current swap file size. Subtracting the minimum required free space (specified in the SWAPPATH= entry in config.sys) will yield the maximum possible size of the swap file at the instant of the query. Once any file activity occurs, however, the amount of free space can change, and it may no longer be possible to provide the amount of swap space indicated by the response to the query. The value returned by QuerySwapSpace therefore cannot be used for overcommit accounting, but merely as a hint of available space.

To avoid the risk of deadlock, excessive stack consumption, and other complications, OS/2 will not support recursive page swapping operations. That is, once a thread has begun a SMSwapIn or SMSwapOut, it may not request another SMSwapIn or SMSwapOut until the operation has completed. Since SwapIn calls are generated by page faults, causing a page fault in this situation will cause the system to halt with an internal error. To avoid causing page faults, the Swap manager and any routines it calls will not be allowed to make memory allocation requests or touch non-fixed memory during SMSwapIn or SMSwapOut calls. They will be allowed to cause faults during other calls. The File System and other portions of the kernel called by the swapper must also be prevented from causing page faults while performing swap I/O. In 286 DOS-boxes, all swapping occurred in the context of one single-threaded swapper process, which was distinguished by the value of the

PTDA variable SwapperProc. This will not work for the page swapper, however, because swapping can occur in the context of any thread in the system. Therefore, a special TCB variable (TCB_fSwapping) will be set to indicate that this thread is in the midst of a swap operation. The File System may use this flag to determine which pool to allocate device driver request packets from, and the Page Manager will use it to determine how to satisfy lock requests, just as SwapperProc was used in v1.0.

Swapping to IFS via FSD

While OS/2 version 1.0 used the FAT based file system inherited from MS-DOS, versions 1.2 and later support the concept of Installable File Systems which are managed by a user supplied File System Drivers (FSDs). It is a requirement that it be possible to swap to an FSD, since it is conceivable that a machine could run with no FAT file system present. In order to achieve this, several requirements have been placed on FSDs that wish to support swapping.

- No allocation of memory during swap operations.

The FSD cannot attempt to allocate memory in support of a swap operation. If it were to do so it could lead to an endless chain of memory allocations causing swap operations causing memory allocations.

- Partial reentrancy required

Since page faults can occur during file I/O operations, the FSD may be required to interrupt those operations to perform swapping I/O. Specifically, the FSD must be capable of being reentered, in the context of the same thread, to satisfy a Read, Write or NewSize request on the swap file. Recursive page faults are not supported, so the FSD need support only one level of recursion.

To help the FSD set up its data structures properly, when the swap file is opened at initialization time the FSD will be informed which SFTs are for the swap file. The FSD may then mark the SFTs for later identification. All I/O using a specially marked swapper SFT must be presumed to be swap I/O, and treated according to the rules listed above.

Data Structures Description

There are two major data structures used by the Swap manager: The SF array and the DF bitmap. To minimize memory usage, especially in the case of swapping not being enabled, these and other data structures will be allocated dynamically.

The major Swap manager data structure is the SF array, with one cell per SF. Each SF cell contains a number of pointer fields used to maintain linked lists, a reference count and a flags word, as shown below.

+-----+ forward link +-----+	(near pointer)
+-----+ backward link +-----+	(near pointer)
+-----+ PF cross link +-----+	(near pointer)
+-----+ reference count +-----+	(unsigned short)
+-----+ flags word +-----+	(16 bits)
+-----+ disk frame index +-----+	(unsigned long)

Swap Frame structure layout

At any point in time cells in the SF array can be partitioned into four groups.

- Those SFs that are currently allocated.

- A doubly linked list of all the free frames (the free list).
- A doubly linked list of the frames mapping pages which are associated with both a PF and a DF (called the idle list).
- Free frames which are beyond the intended new size of the swap file (this is called the shrink list and is non-empty only during a pending shrink of the SF array).

Frames are moved from the free list to the shrink list only when they are encountered in allocation attempts; no active scan of the free list is made. To simplify the problem of detecting when all of the frames past the border have been collected, the shrink list will be kept as a sorted, compressed list, with each cell on the list containing the number of contiguous cells following it, and a pointer to the next block of cells. When the list coalesces into a single block containing all the cells to be freed, the unneeded frames are decommitted.

The SF array will have to be allocated as fixed and non-swappable, but must be reallocatable to match changes in the swap file size. Since under normal circumstances the array will be quite small, it will be allocated sparsely. This means that enough linear address space must be reserved to contain a maximum size array (approximately 16M), but physical memory will be associated only with the portion of the array that is active.

A number of variables are required for overcommit calculations. Foremost is SwapSpace, the total space (in units of pages) that the swapper has committed to provide for the Page Manager. Since this may be less than the actual size of the swap file, the true size must also be stored. The swap file will be grown when the Page Manager requests more swap space than the swap file currently provides. When the Page Manager reduces the amount of swap space required, the Swap manager will not immediately reduce the space it has committed to provide, but instead will wait until all active pages have coalesced to the front of the swap file before resizing the file.

The Swap manager will also need to maintain a number of SFTs for the swap file. Since the FAT file system modifies some fields in the TCB, storage will have to be allocated to preserve those fields around the calls. There are several possible strategies for allocating this memory:

1. Allocate SFTs, etc., from the system as needed

In addition to the obvious speed penalty, this solution would run a risk of deadlock. The system SFT pool is in a BMP segment which may try to grow to satisfy the allocation request, while growth might not be possible until after the page out operation had completed. The same problem would exist with using standard allocation routines for TCB save buffers.

2. Allocate a single swapper SFT at init time

This is a very simple solution, but would enforce unnecessarily strict serialization. Performance can be improved dramatically (especially on non_FAT filesystems) by allowing overlapped swap operations.

3. Allocate one SFT and TCB save area and several request packets per thread in the system.

This allows for the greatest degree of overlap and the least risk of deadlock, but wastes large amounts of memory. With a maximum of 4096 threads in the system, this could take close to 1M ((4096 threads) * (60 bytes/SFT + 80 bytes of TCB variables + 3*32 bytes per request packet)) of fixed memory. At any point in time the vast majority of this space would be unused, since the disk can be saturated with only a few (<10) simultaneous operations. If the total space required grew past 64k, addressability problems would arise, since 16:16 pointers to these structures must be passed to existing 16 bit code. To avoid statically allocating this very large structure in a system with only a few threads, hooks would need to be added to the CreateThread code to grow the pool at thread creation time, adding additional complexity.

4. Allocate from a small fixed size, pool

At init time the swapper will allocate enough memory to hold roughly 8 to 16 SFTs and the corresponding TCB save areas. (The number will be tunable and the final value set by experimentation and measurement.) If the pool is exhausted then further requests will block until one or more of the current operations completes. This does not pose a danger of deadlock because the resources are all owned by threads performing swapping operations, which are guaranteed to complete without causing page faults or requiring more resources.

The small allocation pool is the superior solution. Properly implemented it will run without any risk of deadlock, and will do so without consuming large amounts of memory. The pool will be small enough that simple allocation techniques can be used while still allowing high performance.

Imported Interfaces

- PGInitSwap(Flags)

Used at init time to inform Page Manager what kind of parameters to pass to to swap routines, where the flag fields include:

VPTYPE = address type (virtual 16:16 or physical)
SwapType = is swapping enabled?

- VMAllocMem

Used to allocate linear address space and physical memory for growable data structures, such as the bitmap.

- VMSetMem

Used to grow or shrink sparsely allocated structures.

- SFT file I/O

File System functions used to access the swap file.

- open
- close
- read
- write
- seek
- newsiz
- w_qfinfo

operations fail.

- InternalError(string)

Prints the error string and halts. For use in debugging.

Internal Interfaces

The functions the list below describe the interface between the swap management and I/O layers of the Swap manager.

- SIOSwapOut(DiskFrame, VAddr)

The SIOSwapOut routine writes nPages pages starting at address VAddr to the backing store starting at the location indicated by DiskFrame.

- SIOSwapIn(DiskFrame, VAddr)

Reads the contents of the disk frame indicated by SwapId into the page frame indicated by VAddr.

- SIOResize(nPages)

Changes the size of the backing store to nPages * 4k bytes. Returns an error if the size cannot be changed (e.g., no space left in file system)

- SIOSpaceQuery

Returns an estimate of how large the backing store can be grown. The value returned is a hint rather than a guarantee, and the Swap manager may not actually be able to make this much swap space available should SIOResize be called with the value returned by SIOSpaceQuery.

Design Constraints

To be added.

Design Review

To be performed.

Swap component Implementation

To be added.

Implementation Review

To be performed.

Swap component Appendix

This page is intentionally left blank.

Glossary

backing store

A generic term for the receptacle for swapped out pages. It can refer to a swap file, a swapping partition, etc.

broken frame

A broken DF is one that has caused read or write errors on a previous swap operation. Once this has happened the DF is no longer trusted and is placed on a list of "broken" frames. Broken frames are removed from the allocation pool so that they will not be reused.

page aging thread

A system thread that scans the list of in-memory pages to determine which one(s) should be swapped out. It is possible that a separate thread will not be used, but instead time will be stolen from whatever thread happens to be running. See the Page Manager document for more details.

DF	A 4k block of disk space used to hold a swapped out page, or, alternatively, the integer number used to refer to this block.
SF	A small resident structure used to describe a page of swappable memory. The SF serves as the link between the page table entry, the page frame, and the disk frame.
swap id	A 20 bit value returned by the Swap manager to the Page Manager which is used to locate a swapped out page on the backing store.

Size and Performance Considerations

Data structure memory requirements:

- 20 bytes per swap frame
- Handfull (<100 bytes) of helpers
- One byte TCB variable (TCB_fSwapping)
- small pool of SFTs and TCB save buffers (<= 5k bytes)

Implementation Estimates

	LOCs	Effort	ELOCs	MM
Swap Manager				
initialization	250	0.5	125	0.5
Interpret SWAPPATH= in config.sys, set up data structures such as allocation bitmap, fix up indirect calls				
swap frame allocation	250	0.4	100	0.4
Allocate regions in the swap space to handle swap out requests				
overcommit	200	1.0	200	0.8
Keep track of total memory commitments vs. available swap space, adjust total swap space when necessary.				
freeing, other	200	0.3	50	0.2
Handle swap free requests, other support routines				
File System Swap I/O				
initialization	250	0.4	100	0.4
Create swap file, initialize SFT(s)				
worker routines	250	0.2	50	0.2
SFT I/O package				
interfacing code	150	1.5	225	0.9
Translate SwapMgr requests into SFT I/O calls, error handling, file resizing				
Device Driver Swap I/O				
initialization	250	1.0	250	1.0
Interrogate device driver, initialize				

request packets, etc.				
worker routines	200	1.0	200	0.8
Rough equivalent of DevIOCall2				
interfacing code	150	1.5	225	0.9
Translate SwapMgr requests into device				
driver calls, error handling				
device driver changes	200	1.5	300	1.2
Modify existing disk device driver to				
recognize new partition type and read				
and write operations.				
Total	2350		1825	7.3

Implementation items to be added:

Device Driver Swap I/O		
Changes to FDISK to enable it to recognize	50	1.0
and create new swapping partition type		
File System Swap I/O		
Code to support QuerySwapSpace call	50	1.0

VM component

This page is intentionally left blank.

VM component Architecture

This page is intentionally left blank.

Problem Description/Objectives

OS/2 v 2.0 is Microsoft's and IBM's platform for applications development for the foreseeable future. It must compete with other high-end microcomputer operating systems, such as UNIX, and become the dominant standard in the marketplace. UNIX's major advantage is portability, and it is significant, especially as RISC microprocessors capable of greatly outperforming Intel's 80X86 processor family become available. Though the 80386 and its successors will continue to be strategically important in the immediate future, OS/2 v 2.0 must be designed to be portable to other architectures. This is a major departure from the design philosophy of OS/2 v 1.x, and its impact will be felt by every component of the system, but especially by VM component.

The 80386 will be the first processor to host OS/2 v 2.0. The design of the system must take the fullest advantage of those features of the 80386 that we expect it to have in common with future strategically important processors, while at the same time it should use features specific to the 80386 only to the extent necessary for upwards compatibility with OS/2 v 1.x.

Support of memory objects larger than 64Kb has been desired by application developers for a long time. While the 80386 supports segments larger than 64K, OS/2 v 1.x's memory management scheme is ill-suited to such segments. OS/2 v 1.x support segment-based memory overcommit and swapping. A segment must be swapped in and out of physical memory as a whole. A segment cannot be fragmented; it must occupy one contiguous piece of physical memory. These features prevent the allocation of a segment larger than the physical memory, and

make it very difficult to manipulate large segments. The task of finding or creating a large enough contiguous piece of physical memory to load a segment becomes increasingly difficult as segment sizes grow much beyond the 64K limit present in OS/2 v 1.x. These problems, coupled with the fact that up to eight segments need to be present simultaneously in order to execute an instruction, present insurmountable challenges to a segment-based memory management system. Furthermore, many developers prefer to avoid segmented architecture in favor of a flat address space.

OS/2 v 2.0 must continue to provide the same level of memory sharing functionality that OS/2 v 1.x provides. Dynlink libraries and interprocess communication based upon shared memory areas will continue to be a key feature of the system.

OS/2 v 1.x provides data protection for subsystems using the selector ring architecture of the 80286. Since this method is tied to the segmentation hardware, it is neither portable to other architectures nor appropriate for a flat model system. OS/2 v 2.0 must provide some means of data protection that is portable and is at least as good as that offered by OS/2 v 1.x.

To summarize, the VMM must meet the following objectives:

1. Provide a memory model that is portable to new architectures.
2. Support memory allocations larger than 64Kb. Allow applications to create memory objects that are larger than available physical memory. Provide a memory overcommit algorithm that is compatible with these goals.
3. Support flat address space memory management for new applications, bypassing segmented architecture.
4. Provide for new applications the same level of memory sharing capabilities available in OS/2 v 1.x.
5. Provide a portable means of data protection at least as strong as OS/2 v 1.x's.
6. Support applications written for OS/2 v 1.x.

Solutions/Justification

In terms of memory management, the key features of the 80386 are paged virtual memory support and 32-bit wide segments. While segmentation is a feature that is not commonly found in other families of processors, note that 32-bit wide segments can be used to simulate a flat, 32-bit virtual address space, a very common feature of other processors. On the 80286, the segmentation hardware maps a selector and an offset through the appropriate descriptor table entry to a physical address. On the 80386 with paging enabled, another level of indirection exists between the selector mapping and physical memory: the linear address space, a 32-bit wide flat address space that is mapped through page tables to physical memory. If a selector is created that maps the entire linear address space as code and another is created that maps it as data, and if these selectors are loaded into an application's segment registers and never changed, then for all intents and purposes, an application views the machine not as having a segmented address space, but as having a flat address space. Since this memory model dominates the non-Intel microprocessor market today, we will use it as the model for OS/2 v 2.0.

An application may allocate regions within the flat, 32-bit virtual address space. Such an allocated region is called an object. Conceptually, an object is like a segment in that an object may be treated as a discrete entity for the purposes of certain operations. However, unlike the segmented model, all objects addressable by a process are simultaneously addressable; there is no need to load a segment register with a special value in order to reference a particular object. The use of paging permits allocation of objects not only larger than 64K, but also larger than available physical memory. Paging also allows more efficient swapping and memory overcommit algorithms. Paging eliminates all of the previously mentioned pitfalls of segment-based swapping and swap file fragmentation problems present in OS/2 v 1.x as well.

A flat model system must provide support for objects with shared addresses: objects whose virtual addresses are the same in all the contexts in which they are used. There are two basic kinds of such objects. Shared objects have shared addresses and shared data. Each process sharing the object not only uses the same addresses but also accesses the same physical storage. DLL instance objects are a DLL's per-process data. They must have shared addresses in order for the DLL's code to be shared: if the code is shared, then the code references the same addresses in each context. However, the actual physical storage must be different, since the contents must be allowed to vary from process to process. In OS/2 v 1.x, this sharing was provided at the segment (or selector) level by giving each process its own LDT. Certain LDT selectors were reserved globally for use by DLLs (as well as other shared memory). OS/2 v 2.0 does not have this level of indirection for flat model applications; thus, it cannot be used to provide virtual address sharing. This sharing must be provided by the Page Manager at the page table level by giving each process its own set of page tables. Just as OS/2 v 1.x partitioned the LDT into a set of private selectors (i.e., selectors reusable on a per-process basis) and shared selectors, OS/2 v 2.0 will provide the same level of functionality by giving each process its own private arena and by having another arena shared among all processes. An arena is a contiguous subset of the processor's virtual address space. A private arena is analogous to the set of private selectors, and the shared arena is analogous to the set of shared selectors.

In OS/2 v 1.x, shared and private selectors were interleaved in the LDT, no problem since there is no direct relation between a selector and the location or size of the segment to which it refers. Obviously, the same cannot be said for 0:32 addresses, since they define an object's location and size in the virtual address space. The simplest way to partition the address space into private and shared areas is to put objects with private addresses at one end and objects with shared addresses at the other and have them grow toward each other, much as one would manage two stacks at the same time. OS/2 v 2.0 will begin the private arenas at the low end of the users' address space and put the shared arena at the high end. They will grow toward each other, but may never overlap. The boundary between the two areas will float according to the requirements of the applications running in the system, but there will be minimum and maximum values between which the boundary will be constrained to remain.

Support of both 0:32 and 16:16 addressing formats in the same process requires some means of translating a 0:32 address into a 16:16

address and vice versa. In OS/2 v 2.0's memory model, a 0:32 address is the same as the linear address. Thus, converting a 16:16 address to a 0:32 is accomplished by obtaining the segment's base linear address from the descriptor table entry specified by the selector and adding the offset to it. Translating from 0:32 to 16:16, conversely, requires the existence of a descriptor table entry that maps the appropriate region of the linear address space. OS/2 v 2.0 could provide APIs to perform these translations. The former would require a table lookup and an addition, a relatively cheap operation except for the overhead of calling the system. The latter, however, would be more difficult: a descriptor would have to be allocated and initialized, and the system would have to provide another API to destroy the mapping when it was no longer needed. This solution to the translation problem is inefficient.

What if there were a simple relationship between a 0:32 address and its corresponding 16:16 address so that each could be easily derived from the other using a simple sequence of machine instructions with no system intervention? Such a relationship would be efficient and easy to use. In fact, it can be done. In OS/2 v 1.x the maximum amount of memory a process can own is limited by the amount of memory its LDT can map. An LDT can have 8192 descriptors, each mapping up to 64Kb of memory. Therefore the maximum amount of memory an application can have is (8192*64Kb =) 512 Megabytes. This amount excludes system owned memory mapped by GDT, but includes all shared memory allocated by processes in the system. In OS/2 v 2.0, this limit will be retained for each process in order to have one-to-one mapping between the two addressing formats. For segmented applications, the base address of every segment will be equal to the 13 high-order bits of the selector (the index of the corresponding descriptor in the LDT) times 64K. The low 3 bits in a selector determine the table and ring protection levels, and therefore are not indicative of distinct memory addresses. Thus, conversion from a 16:16 (selector:offset) address to a 0:32 address (linaddr) will be done via the formula:

$$\text{linaddr} = (\text{selector} \gg 3) \ll 16 + \text{offset}$$

When a 0:32 application allocates an object, either statically in its load image or dynamically at runtime, it has the option of specifying whether the object is to be tiled. A tiled object is one that is allocated on a 64Kb boundary and has the appropriate descriptors in the LDT mapping each 64Kb increment of the object. Thus, it is possible to construct a valid 16:16 address for any byte within a tiled object by using the inverse of the formula shown above, namely:

```
offset = linaddr : 0xFFFF
selector = ((linaddr >> 16) << 3) | 4 | CPL
```

In future versions of the system, untiled objects will be allocated above 512M and below 4G. Such objects will, of course, be inaccessible from true 16:16 code.

Note that tiling is not a perfect solution. New applications that wish to use existing 16:16 code must be careful of the tiling boundaries within objects. In particular, a single logical entity, such as a buffer or a name string, must not cross a 64K boundary if it is to be used by 16:16 code since the entity will not be completely accessible using a single selector. Language tools must provide support to insure that such mistakes are not made.

One side effect of page based memory management is that all memory allocation will be done with page size granularity. Every memory object, including the segments created by old APIs, will occupy at least one physical page. This may cause inefficient use of memory in applications where there are a lot of small segments (much smaller than a page). Application programmers must be educated about this. This inefficient memory usage will diminish as programmers combine their small segments into larger memory objects of one or more pages.

Another side effect is loss of byte granular protection. In OS/2 v 1.x any attempt to read or write beyond a segment's limit produced GP faults. This will no longer be true in the flat linear model. Application developers can no longer rely on GP faults to detect their bugs caused by read/write attempts beyond the memory object's limit.

New API calls allowing the applications to manipulate 0:32 objects must be provided. These new applications must also be able to communicate with existing dynalinks and device drivers in the 16:16 bit format.

One compatibility problem that arises with page based, discontinuous segments is support of DMA. Current OS/2 device drivers access physical memory directly, bypassing the VMM. They also assume that segments in memory are physically contiguous. With support from the Page Manager, the VMM should be able to provide contiguity to memory objects involved in DMA.

Architectural Review

To be performed.

VM component Design

This page is intentionally left blank.

Design Overview

The VMM can be divided into the following subcomponents:

1. Data Structure Management
 - a. VMARs
 - b. VMOBs
 - c. VMCOs
 - d. Mapping a Virtual Address to an VMOB
 - e. Semaphore Management
2. Object Management
 - a. Private Arena Creation and Deletion
 - b. Allocation
 - c. Reallocation
 - d. Freeing
 - e. Protecting DLL shared data.
3. Unnamed Object Sharing and Attachment
4. Named Shared Object Management
5. Sparse Object Support
6. Object Locking and Unlocking
7. Aliases
8. Discardable Object Management
9. Kernel Data Management
 - a. BMP
 - b. RMP
 - c. Kernel Heaps
10. Initialization
11. Interfaces
 - a. API
 - b. DevHlp
 - c. FSD

The following sections discuss each of the subcomponents in detail.

Data Structure Management

1. VMARs

An arena is represented by a doubly linked circular list of VMARs sorted in ascending order of base virtual address. The first VMAR in each arena is a sentinel used to mark both the beginning and the end of the arena. It also contains the size of the arena. Each additional VMAR corresponds to an object, or local alias (see "Aliases"). In addition to the address, and links, an VMAR contains the size of the object, an additional VMAR link whose purpose is described below, the handle of the associated VMOB (if any), an indicator of which arena contains the VMAR, and flags. Free regions within an arena are represented by consecutive objects that are not contiguous (i.e., the base plus size of the *n*th object is less than the base of the *n*+1st object).

There are several possibilities for managing the pool of VMARs:

- Dynamically sizable table with 16-bit indices

Using this method, a maximum of approximately 64K VMARs could be created. Additional data is needed to decide if this number is sufficiently large. The advantage is that the VMARs themselves are each six bytes smaller, since the three link fields to other VMARs are each half the size of a full pointer. The table could be managed by the BMP32 package.

- Dynamically sizable pool with 32-bit pointers

This method removes the 64K limit and simplifies addressing at the cost of increasing the amount of memory

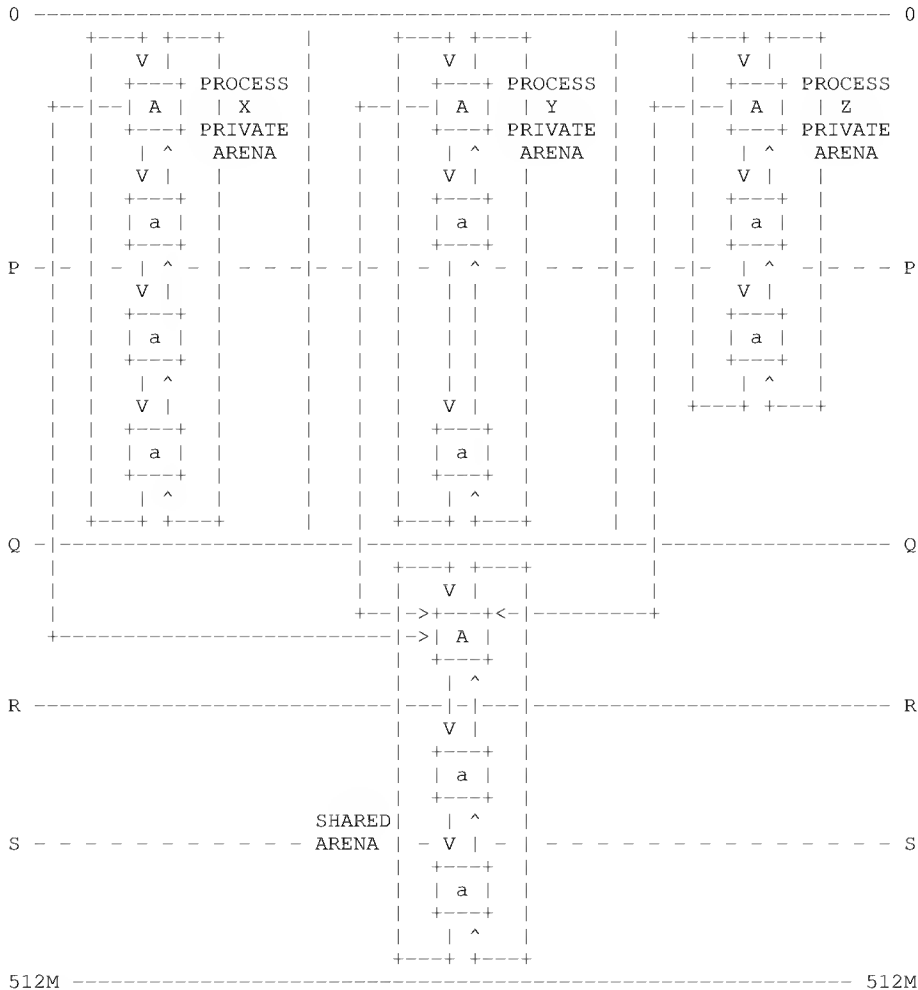
consumed for each VMAR.

- Allocation from a kernel heap using 32-bit pointers

This method is the cheapest to implement because it requires no special initialization code, a minor advantage. It is the most expensive in terms of space consumed, because the VMARs must use pointers since they won't be contiguous, and because each VMAR will contain the header used by the kernel heap manager.

The first method is preferable if there is no need for more than 64K VMARs. Otherwise, the second method is preferable.

The figure shows the arenas in a system running three processes.



Arena Layout in System Running Three Processes

Each process has its own private arena list that describes the objects in the virtual address space between virtual address 0 and virtual address Q. These addresses are "local", or specific to each process. The first VMAR in a private arena is always a Start Sentinel with virtual address 0 and size 0. The last VMAR in each private arena points to the Start Sentinel for that arena. Each private arena Start Sentinel points to the Boundary Sentinel. The Boundary Sentinel is a VMAR having address Q and size R-Q. A flag in the VMAR distinguishes the Boundary Sentinel from other VMARs. By varying the address and size in the Boundary Sentinel, one can effectively adjust Q and R, thus varying the boundary between the end of all private arenas and the beginning of the shared arena. R-Q can never be less than zero. Q can never be less than P, the minimum allowable shared address (64M). R can never be greater than S, the maximum allowable private address (448M - 1). All processes share the shared arena list that describes the objects in the linear address space between Q and 512M. These addresses are global. In the first release, nothing is mapped into the linear address space between 512M and the start of the system arena. Objects in the system arena are mapped into every process' address space. The actual address at which the system starts is determined during System Initialization. Each process has its own set of page tables mapping the addresses from 0 to 512M, even for the shared arena. Separate page tables for the shared arena permit instance data and selective access to shared data. There is one global set of page tables for the system arena. This area is protected so that only the kernel can access objects in it.

2. VMOBs

An VMOB contains an object's flags, its owner i.d., its MTE handle (if it is defined by a load module), its semaphore, and its lock counts, as well as a pointer to the list of arena records associated with the object. An VMOB is most commonly referenced by its handle. A handle is simply an index into the system's table of VMOBs. A handle is 16 bits wide, implying a maximum of 64K-1 objects in the system at any time (handle 0 is the null handle).

Handles serve a special purpose in OS/2. They are used as owner identifiers for objects. There are three kinds of owners:

a. Special owners

Special owners are the system, device drivers, and dynamic shared memory. Some relatively small number of handles are reserved in the handle space. These handles do not have VMOBs associated with them. They are used exclusively as owner i.d.s. Given that there are n such i.d.s, then the maximum number of objects in the system is actually $64K-n-1$.

b. Processes

A process owns any memory objects that are private to that process. The handle of the process' PTDA object is used as the owner i.d.

c. MTEs

MTEs "own" any shared memory objects that they define. The handle of the MTE is used as the owner i.d.

Recall that objects are allocated with page granularity. A PTDA is just over 1K bytes long, and most MTEs are substantially smaller than 4K. Thus, allocating each PTDA and MTE as a separate object is wasteful of space. It would be much more space-efficient to manage PTDA's using BMP32 and MTE's using the kernel heap, (MTE's cannot be managed by BMP32 since they have differing sizes). However, we need each PTDA and MTE to have its own unique identifier drawn from the same name space. Enter the pseudo-object. A pseudo-object has a handle just as a regular object does. That handle maps to an VMOB, but the VMOB for a pseudo-handle contains a flag that distinguishes it from an ordinary one, and it contains the virtual address of the pseudo-object which is actually a part of a regular object in the system arena. Semaphores in pseudo objects' records are also useful for serializing the changes made to the pseudo objects themselves. PTDA semaphores are used for serializing certain actions that must not occur concurrently within the same context, like task termination and giving/gaining access to shared memory.

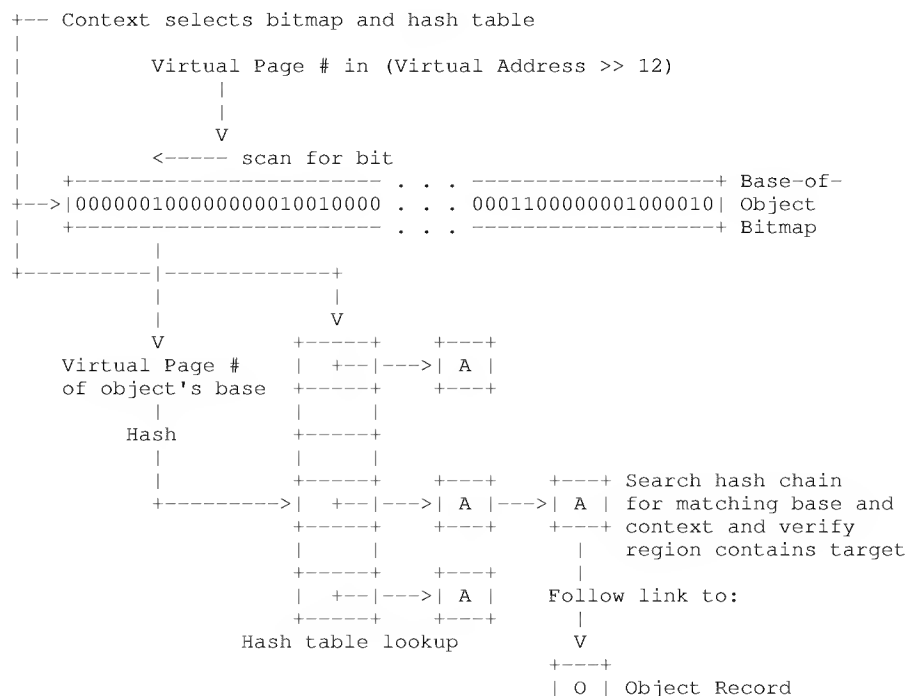
The VMOB table is managed by BMP32.

3. VMCOs

A shared object in the shared arena has one VMAR and one VMOB associated with it. However, this object may be mapped into many contexts. A means of identifying those contexts is needed. For such an object, the context field in the VMAR is not the ptda handle, but a link to a VMCO. A VMCO is simply a record containing a link to another VMCO, a ptda handle and flags. Since the link field in the VMAR is 16 bits, VMCOs are allocated from a table and referenced by index. Thus, there may be a maximum of 64K-1 VMCOs in the system. Since these records are used only for shared objects in the shared arena, this limit should not be a problem. The table of VMCOs is managed by BMP32.

4. Mapping a Virtual Address to an VMOB

In most of the interfaces exported by the VMM, the caller specifies the object on which he wishes to operate by specifying a virtual address within that object and a context (i.e., a handle to a process that can address the object). From these parameters, the VMM needs to be able to find the VMOB typically so that the operation can be serialized properly. The figure illustrates the mapping of a virtual address and context to an VMOB.



Mapping a Virtual Address to an VMOB

Associated with each arena is a bitmap and a hash table. There is one bit for each page of virtual address space in the associated arena (for tiled arenas, one bit represents 64K of address space). A bit is set to one if the corresponding virtual page is the first page in some object. The mapping proceeds as follows:

- a. Shift the virtual address right twelve bits to get a virtual page number.
- b. Using the virtual page number and the context indicator, determine which arena contains the virtual address, and find the associated bitmap.
- c. Use the virtual page number (scaled) as an index into the bitmap. Starting from the given bit, search the bitmap in the direction of descending virtual page numbers until a bit that is set is found.
- d. Hash the virtual page number corresponding to the set bit, and search the appropriate chain of VMARs in the hash table looking for a matching virtual page number.
- e. When a match is found, verify that the object is large enough to contain the original page number.
- f. Find the VMOB to which the VMAR points. This may require scanning a list of VMOBs to match the context indicator for instance data in the shared arena, or it may require scanning a context list for shared data in the shared arena (more on these lists later).

5. Object Semaphore Management

Certain operations on memory objects need to be serialized. Thus, each VMOB contains an exclusive semaphore. Only one thread at a time may own the semaphore for a particular object, but that thread may request ownership multiple times. The semaphore will be released only when the owning thread has cleared the semaphore as many times as it has set it. The routine to wait on a semaphore has several options which can be combined in any fashion:

- Obtain ownership

The default is not to take the semaphore, but simply to check if it is available. Setting this flag causes ownership to be granted if/when the semaphore is available.
- Wait/No wait/Error wait

This option controls what happens when the semaphore is unavailable. There are three possibilities. The default is to block until the semaphore is available. It is also possible to return an error immediately if the semaphore is unavailable, or to block and then return an error. This last option is used commonly with user-level objects. In a multithreaded environment, one thread cannot be sure that the object of interest will still exist after blocking, since another thread of the same process may have freed it. Thus, if the calling thread blocks, then when it awakes, it must verify that the object still exists before trying to take the semaphore.
- Wait for short-term locks to clear

The default is to ignore short-term locks. If the wait option is set, and there are outstanding short-term locks, an indefinite wait normally occurs until all short term locks are clear. There are two exceptions to this rule. First, the wait may be interrupted by a signal. Second, the wait will be timed out after 10 seconds if the only short-term locks outstanding have been imposed by device drivers (using the device help interface). This safeguard helps to prevent a device driver that fails to release a lock from impacting the system. Signals and timeouts can be masked.
- Ignore signals and timeouts

The default is to honor signals and/or timeouts occurring during a wait to take an object's semaphore, or for locks to be released. If the ignore option is specified, signals and/or timeouts will be effectively ignored; a wakeup from a wait caused by a signal or timeout will result in a retry and repeat wait if necessary.

The specific operations that require serialization are discussed in the following sections.

Object Management

1. Private Arena Creation and Deletion

A private arena is created as follows:

For both the tiled and untiled arenas, do the following:

Allocate an VMAR to serve as a Begin Sentinel marking the beginning of the arena.

Link this Begin Sentinel to itself to signify an empty arena.

Set the arena end to the minimum allowed shared address.

Link the Begin Sentinel to the shared arena sentinel.

Store the handle of the Begin Sentinel in the PTDA.

Note: The untiled arena will not be supported in OS/2 v 2.0; it will be supported in a later release.

A private arena may not be deleted until it is empty. Deleting it is simply a matter of freeing the Begin Sentinel and removing the link to it from the PTDA.

2. Allocation

A request to allocate a new memory object is not prone to most of the serialization problems that plague other VMM operations since the new memory object is not visible to the system until after the allocation operation is complete. It is necessary to co-ordinate allocations of memory objects that result in arena size changes. A reservation of virtual address space in the shared arena must be protected against encroachment from a concurrent reservation in a private arena that causes the private arena to grow and possibly require the shared arena to contract. Each arena has a semaphore which is used to serialize arena size changes. All requests other than allocations require that the caller specify a particular object to be acted upon. Thus, situations arise in which two different callers may try to operate on the same object at the same time. However, the identity of an object that is being created is not externally visible until the allocation is successfully completed. Allocation requests can block, however, so it is possible for two or more allocations to be in progress at one time. Therefore, the VMM must insure that its internal data structures are in a consistent state.

Under OS/2 v 1.x, overcommitment of physical memory is supported by swapping whole objects and compacting the physical arena to reclaim space rendered unusable by fragmentation. OS/2 v 1.x supports allocation of the following types of segments:

fixed

A segment that can be neither swapped nor moved; it resides at the same location in physical memory for its entire life.

movable

A segment that can be moved by the compactor if it is not locked; however, it is always resident in memory.

swappable

A segment that can be swapped out to the swap device if it is not locked and subsequently faulted back in, perhaps at a different location in physical memory.

discardable

A segment that can be thrown away if it is not locked; its contents can either be reloaded from an EXE file or recreated by the application that created it (so-called API discardable objects). Note that a segment is

always contiguous in physical memory, since paging is not used by OS/2 v 1.x.

In OS/2 v 2.0, memory type is maintained on a per-page basis, rather than a per-object basis, and the client has the ability to change the type on a per-page basis. Thus, from the VMM's point of view, there is little meaning in talking about the type of an object the way one talks about the type of a segment. It is possible for an object to have regions of different types simultaneously. However, when an object is first allocated, all of its pages have the same type. The types supported by OS/2 v 2.0 are as follows:

fixed

All pages can be neither swapped nor moved; they are not necessarily contiguous in physical memory unless contiguity is explicitly requested.

resident

Same as fixed.

swappable

All pages are swappable.

discardable

All pages are discardable; if a page is discarded, its contents can be reloaded from an EXE file. There are no API-discardable objects under OS/2 v 2.0.

invalid

All pages are invalid. Pages must be committed to regions of the object via a separate call to the VMM before the object can be

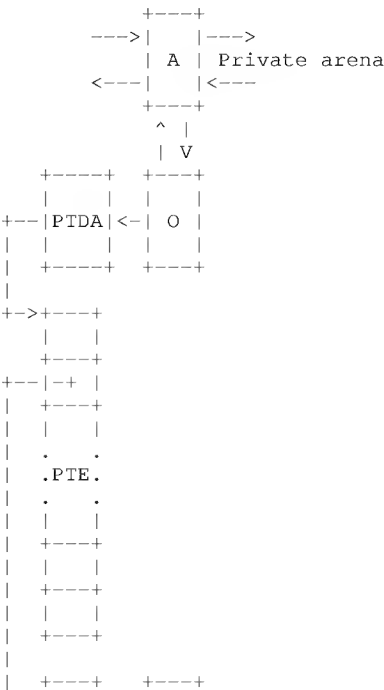
referenced. See the discussion of "Sparse Object Support" below.
 Note that the virtual address of an object never changes. Objects do not move in the virtual address space.

There are four basic kinds of objects that can be allocated:

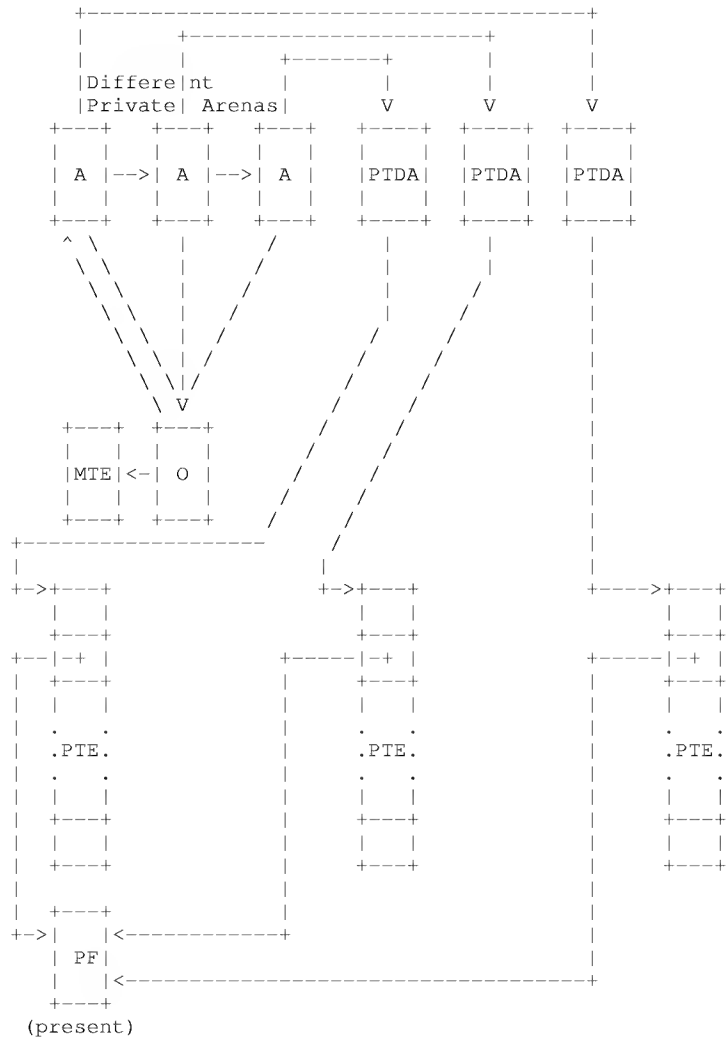
- a. Private virtual address, private contents
 EXE private writable data and dynamically allocated private data.
- b. Private virtual address, shared contents
 EXE code, EXE read-only data, and EXE shared writable data.
- c. Shared virtual address, private contents
 DLL instance (private) writable data.
- d. Shared virtual address, shared contents
 DLL code, DLL read-only data, DLL shared writable data, named shared memory, and unnamed shared memory.

The following figures show how the VMM data structures are linked with the other data structures in the system in these four most common cases. Some symbol definitions:

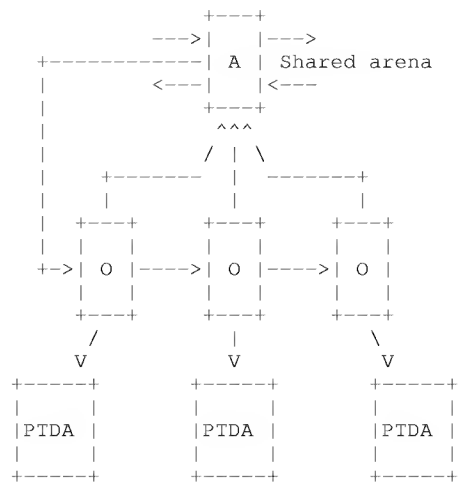
A	An VMAR.
O	An VMOB.
PTDA	A Per-Task Data Area, repository for all sorts of per-process information, including two VMAR handles and linkage to the process' page tables.
PTE	A Page Table, used to contain information used both by the Page Manager and the processor.
PF	A Physical Page Frame structure, a Page Manager data structure that contains information about each page of primary storage in the system.
SF	A Swap Frame structure, a Page Manager/Swap manager data structure that contains information about each page of secondary storage in the system.
CTXT	A VMCO.

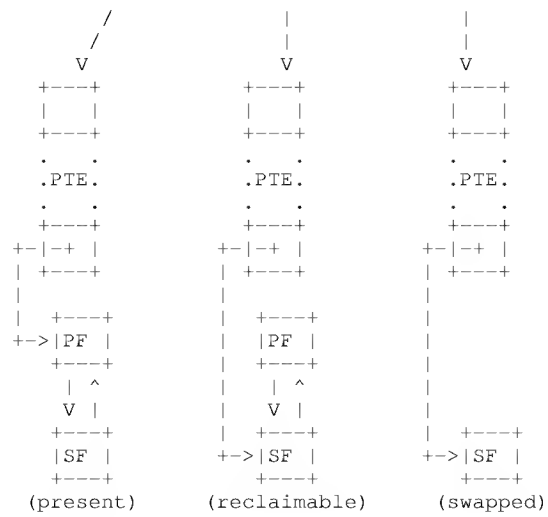


Private Writable EXE Data

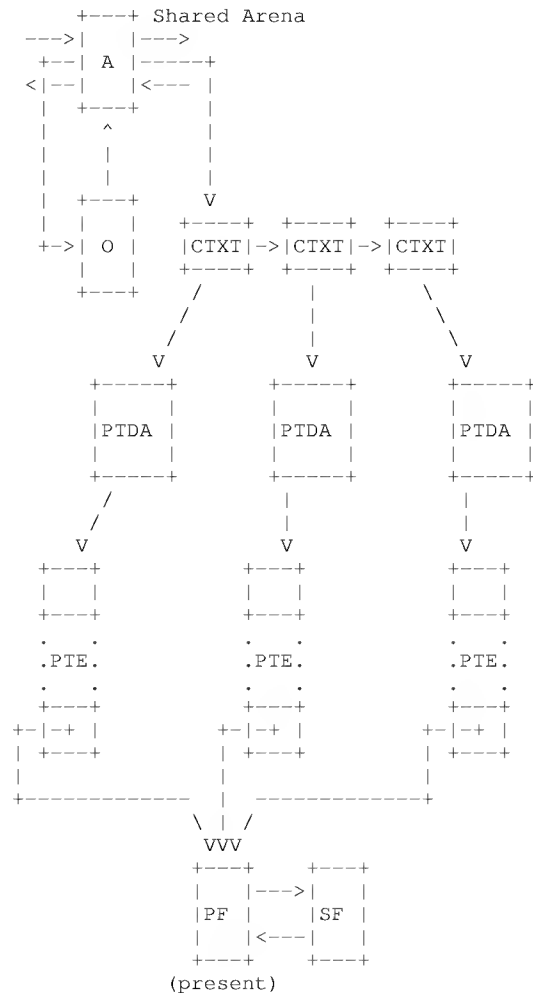


EXE Code





DLL Instance Writable Data



DLL Shared Writable Data

A request to allocate a memory object specifies the following parameters:

reservation size

Desired maximum size in bytes. This size equals the amount of virtual address space to be assigned to the object.

initial committed size

	Size of the region of the object (beginning from its start) for which backing storage (memory or media) is to be initially assigned. Additional regions may be committed later.
flags	Type of memory wanted, arena specifier (system, shared, private), shared-contents flag, address specifier (specific, above-specific, or any), access rights (read, write, execute, user), tiling flag, alignment information (page-, 64K-, or 4M-aligned), storage-decommit-allowed flag, givable flag, and gettable flag.
PTDA handle	The object handle of the PTDA for the process into whose address space the object is mapped. Also specifies the arena if the object requires private addresses.
owner	The owner of the object.
MTE handle	Handle of MTE that defined the object for objects defined by loaded modules
retinfo	Pointer to a buffer in which to return the address of the object and its object record handle and selector (if any) if the allocation succeeds. If allocation is requested at or above a specific virtual address, or a specific selector is to be mapped to the object, the address and/or selector to be used are placed in this buffer prior to calling the allocation function.
block number/physical address	Starting block number in load image for discardable objects created by the loader. The VMM passes this parameter to the Page Manager without examining it. This location is also used to contain a specific physical start address where required for non-loader objects. The basic allocation algorithm is as follows:

Set the requested size to the nearest multiple of the page size greater than or equal to the requested size.

Determine the arena from which to allocate. Use the system arena if the system address space is requested; use the shared arena if the shared address space is requested; use the arena selected by the PTDA handle if the private address space is requested;

Call the VMM subroutine that allocates virtual memory.

Call the Page Manager to obtain storage for the virtual memory.

Call the Selector Manager function to allocate/map a selector if needed. Error recovery will require undoing the steps completed successfully at the time of the error.

3. Reallocation

Reallocation is supported strictly for compatibility with OS/2 v 1.x. The Selector Manager simulates reallocation of 16:16 segments with the help of the VMM. For any allocation request in the tilable region, the VMM will allocate the virtual space with 64K granularity and 64K alignment; the physical memory will be allocated with page granularity. Thus, an object in the tiled area may be sparse implicitly. The limit maintained by the Selector Manager in the LDT descriptor for a 16:16 segment is the to-the-byte limit requested by the user, just as it is under OS/2 v 1.x. To simulate a grow request, the Selector Manager commits pages at the end of the segment valid using the VMSetMem function. Similarly, to simulate a shrink request, the Selector Manager decommits pages at the end of the segment using VMSetMem. From the VMM's point of view, the size of the object never changes. The Selector Manager adjusts the limit in the LDT descriptor as appropriate.

Reallocation for 0:32 objects is not permitted for the following reasons:

- a. Growing an object is unreliable, since the base virtual address of an object cannot be changed.
- b. Growing DLL instance data requires complex data structures:
 - The arena record must track the size of the largest instance, since the virtual space has to be reserved in all contexts.
 - The sizes of individual instances are needed for Page Manager operations; in particular, to distinguish growth (which is allowed) from shrinking (which is not), and to allow growth to behave as it does for private address space objects (when the object is sparse at the end, there is no way to distinguish its virtual end from its physical end without maintaining a per-instance size).

c. Growing EXE shared data requires complex processing:

- The VMM must verify that the necessary virtual space is available in ALL of the contexts in which the object is mapped.
- The Page Manager must be able to allocate enough page tables for ALL of the contexts in which the object is mapped as well as pages for the grown object.
- Failure to meet either of these conditions means the entire growth operation fails and any work done in a particular context must be undone.

Given the technical difficulties with growing DLL instance data and growing EXE shared data, and given the severe restriction on usefulness that growing an object implies, we will not support reallocation for 0:32 objects. Use of the sparse object facility allows the same basic effect to be achieved for most cases.

In some future version, we may wish to add a virtual move facility. Such a facility would allow the user to move the data from a valid area of one object to a sparse area of another object by editing page tables. This facility could be used to simulate reallocation of objects for clients that do not depend on the constancy of the base address of an object.

4. Freeing objects

Freeing an object from a given process means making it inaccessible to that process and, if the object is no longer needed by any other process, releasing some or all of the object's resources (virtual address space, backing storage, selectors). The VMM can detect from its data structures how many processes can access an object. Except for resource objects (see below) a free operation will result in the release of an object's resources when the number of processes with access to the object reduces to zero.

Additional rules govern the effect of a free operation on certain types and states of an object. These rules affect the scope of certain API calls, the ability to reload DLL objects, semaphore management and objects having aliases or locks.

a. Freeing DLL objects

The shared arena virtual address space must remain reserved in case a subsequent attach to the DLL module defining the object is made. Since DLL objects may be referenced by fixups from other objects, and since fixups are made at object load time only, a reload of a DLL object must restore object addressability to the original virtual addresses so long as the module is still referenced. For this reason, the free function retains the virtual address space for a freed DLL object until called explicitly from the Program Loader garbage collection function to free them. The backing storage of a DLL object need not be retained for this purpose, however.

b. Freeing resource objects

A resource object's virtual address space and backing storage must be retained even if the number of processes referencing the object is apparently zero, until the resource reference count (the number of outstanding DOSGETRESOURCE calls) has also reached zero. The free function detects this by calling the Program Loader (function LDRFreeObject) whenever a free operation on a resource object is being performed. The resource reference count is decremented and the LDRFreeObject function returns an error if the resource object is still in use. In this case, the free operation terminates without error and the resource object remains addressable by the target context. The LDRFreeObject function is also called when freeing the last context of any shared content loader object is being freed from the object, so that the Program Loader can update the MTE for the object to indicate that the object is no longer loaded. Note that the LDRFreeObject function updates the Program Loader data structures and hence requires the Program Loader semaphore. to be held.

c. Freeing named shared objects

A named shared object is guaranteed to be addressable by any thread of a process during the time that a DOSGETNAMEDSHAREDMEM call is outstanding. To protect named shared objects from being freed prematurely two reference counts are maintained. First, a per-process reference count (count of outstanding DOSGETNAMEDSHAREDMEM calls made by threads of the process) is maintained. A named shared object cannot be freed from a process until the per-process reference count goes to zero. Second, a global reference count (count of processes attached to be object) is maintained. Although this count could be computed by scanning the existing data structures of the VMM (arena, object and context records), there would be a performance hit. When a DOSFREEMEM call is made to free a named shared object, the per-process reference count is decremented. If this count reaches zero, the object becomes unaddressable by the current process, and the global reference count is decremented. If the global reference count also reaches zero, the object's resources are freed.

d. Freeing objects with aliases

To ensure correct operation of the system, a free operation on an object will not proceed until all aliases to the object (except for CS aliases) have been freed. For DosDebug aliases, the free operation will notify the debugging process for each DosDebug alias that exists, and wait for the debugging process to free the alias. This mechanism ensures that free operations performed by a debuggee process do not cause improper operation of a debugger process. For other aliases, except CS aliases, a free operation on the object's non-aliased address and context results in a cascade of free operations on each of these aliases. The original free operation waits for all of these frees to complete before proceeding. To retain compatibility with existing versions of OS/2, CS aliases can only be freed by an explicit free operation on the alias's virtual address and context.

e. Deadlocks and use of Semaphores during a free operation.

A thread attempting to free an object must wait for both the object's semaphore and the semaphore of the arena relevant to the given address and context to be available before proceeding. Additionally, a free operation on an object owned by an MTE requires ownership of the Program Loader semaphore for the duration of a call to the Program Loader to determine if the object is a resource object and if it can be freed. Similarly, a free operation on named shared memory object requires ownership of the named shared memory semaphore and a call to the named shared memory manager to determine if the object can be freed.

To prevent deadlocks, the semaphore hierarchy must be observed by a free operation. This means that semaphores lower in the hierarchy (such as an object's handle semaphore) must not be held while waiting on a higher semaphore (such as the loader semaphore). Also, since a free operation may yield at various points before proceeding, the state of the object and its associated arena may change at any of these yields. The free operation must therefore have the capability to relinquish semaphores prior to certain semaphore waits and restart the operation from the beginning after yields until all semaphores are acquired and all actions relating to aliases (see above) are complete. the semaphore hierarchy is preserved.

f. Freeing objects with locks.

Short-term locks on an object are treated differently by a free operation depending on whether they originate from a device driver request (a devhlp request) or from the kernel itself. Unless interrupted by a signal, a free operation will wait indefinitely for locks to clear if there is at least one kernel-originated short-term lock present. If, however, the only short-term locks are devhlp locks, waiting is limited to 10.seconds. Waiting for short term locks occurs regardless of the context or address space through which the locks have been imposed.

If an object has long-term locks, a free operation will only wait for them to clear if the object is long_term locked via the address range and context specified to the free operation. If an object is long-term locked only through address ranges or contexts different from that specified, the operation will fail. The free function checks the relevance of the long term locks to the given address range and context by consulting a small database containing entries of the form <arena record handle, PTDA handle>, each entry signifying that a long-term lock is imposed through the address range mapped by the arena record in the context given by the PTDA handle. In this way, locks through aliases and other address ranges/contexts are protected from being undermined by a free operation.

g. Freeing objects during task termination/new task abort.

Private arena virtual address space pre-reserved for the loading of a huge object may only be freed at this time. A special task-termination flag is passed to the free function to indicate that the free operation is being performed within task termination.

h. Freeing objects during Program Loader garbage collection.

Shared arena virtual address space retained for a future attachment to a DLL module in which some objects have been freed from all contexts, or shared address space pre-reserved for loading a huge object may only be freed at this time. A special flag is passed to the free operation indicating that the free is being performed during garbage collection. Any shared data or instance data object associated with the address space must have been freed from all contexts by prior calls to the free function.

i. Freeing objects from a VDM.

VDM's are allowed to free any region of their own virtual address space without regard to object boundaries.

j. Calling the free routine

The parameters to the free routine are:

vaddr

The base virtual address of the object.

PTDA handle

The object handle of the PTDA for the process into whose address space the object is mapped.

flags

Identifies whether special action is to be taken.

k. Basic algorithm for freeing an object

The basic algorithm for freeing an object is as follows:

Get information about the object from its virtual address and context.

Wait for the object's semaphore to be available and for all short-term locks to be removed. If it was necessary to block, then restart the free operation.

Obtain ownership of the object's semaphore.

Obtain ownership of the semaphore for the arena containing the base address in the given context.

If there are debugger aliases to the object and we are freeing it in the debuggee's context, then notify the debugger to free the aliases, free the semaphore, block waiting for the debugger to comply, and restart the free operation upon awakening.

If there are other (non debug) aliases to the object and we are freeing the main object, free all aliases (except CS aliases) created in the target context. Freeing the aliases is accomplished by calling the free routine recursively for every alias to be freed.

If the object is a resource object, or we are freeing the last context of any shared content Program Loader object, then, if the Program Loader semaphore is available, take it and call the Program Loader to decrement resource reference count and/or zeroise handle in OTE to indicate that object is no longer loaded. If Program Loader indicates that object is a busy resource object, terminate free operation with no error after releasing semaphores. If the Program Loader semaphore was busy, release semaphores and restart the free operation.

If the object is a named shared memory object, take the named shared memory semaphore then decrement the reference count for this process. If the count is still non-zero, then release the semaphore and return. If it is a task termination request, clear the reference count, and free named shared memory local record associated with the context.

Call the Page Manager to free the object's mapping to physical storage (if any). This may cause page tables and pages to be freed if they are no longer referenced. The Page Manager may block while doing this.

If the object has selectors associated with it, then call out to the Selector Manager to free them.

If freeing an alias, free the corresponding alias record.

If the object is in a private arena (see the figures), then

Remove the VMAR from the VMOB's VMAR chain and free it.

If the VMOB's VMAR chain is empty (i.e., there are no more references to the object), then free the VMOB. Else (the object is in the shared arena) if the object is instance data (see the figure), then

Remove the VMOB from the VMAR's chain of VMOBs. If the chain is empty, then

Free the VMAR.

Free the VMOB. Else (the object is shared data in the shared arena; see The figure)

Delete the appropriate VMCO from the Context List.

If the Context List is empty, then

Free the VMAR.

Free the VMOB.

5. Protecting DLL shared data

As described under the section "Memory Protection Architecture", the capability to protect DLL writeable data is enabled by default. If enabled, the top end of the shared arena address space is available for protected allocations of DLL global data and run-time shared data only. The protected and unprotected parts of the shared arena are separated by a regular arena record that is created during VMM initialization. The boundary virtual address separating the two regions remains constant and is not configurable. If there is not enough memory available in the protected region to fulfil an allocation request, an allocation will be attempted from the unprotected region. An allocation will not cross the boundary between the regions.

6. Unnamed Object Sharing and Attachment

7. Named Shared Object Management

8. Sparse Object Support

A sparse object is a writable object whose virtual size can be larger than the amount of physical storage (memory or swap space) committed to it. There are two ways to create a sparse object. It is possible to create a writable object for which only the virtual memory and page tables are allocated. One may call the VMM to commit pages to the object as they are needed wherever they are needed. One may also create a writable object that has a full storage commitment, and then relinquish pages of physical storage when they are no longer needed. Pages to which storage has been committed are known as valid pages (because references to such pages are valid). The valid pages in a sparse object need not be contiguous--a sparse object may have "holes" in it.

There are some restrictions:

- One may not relinquish valid pages from a shared, writable object unless the object was allocated as decommitable.
- Pages for which storage has not been committed may not be referenced. If an application attempts to do so, the resulting page fault will be converted into a General Protection violation, terminating the process. If the kernel attempts it, the resulting page fault will cause an internal error.
- The system will fail a request to commit pages to a region of an object if there is insufficient memory available at the time of the request.

Sparse objects are useful for applications managing very large sparse arrays, and for the system in managing data structures whose maximum sizes are known but whose actual sizes vary as the system operates (e.g. page tables, LDTs, etc.).

Sparse object support is actually a specific use of a more general facility: changing the type of storage associated with a region of a memory object. This facility is also needed for other reasons:

- The Program Loader needs to be able to make regions that were initially allocated as discardable swappable instead.
- FSDs need to be able to change the type of regions from swappable to resident (i.e., unswappable).

The figure lists the valid type transitions and their respective clients.

Old	New	Clients	Partial pages
----	----	-----	-----
Invalid	Swappable	App.s, Kernel	included
Invalid	Resident	Kernel	included
Invalid	Fixed	Kernel	included
Discardable	Swappable	Loader	included
Swappable	Swappable	App.s, Kernel	included
Swappable	Resident	FSDs	included
Invalid	Invalid	App.s, Kernel	excluded
Swappable	Invalid	App.s, Kernel	excluded
Resident	Invalid	Kernel	excluded
Fixed	Invalid	Kernel	excluded

Valid Type Transitions for Memory Regions

Since the VMM's clients are for the most part ignorant of paging issues, a region in an object is specified in terms of a virtual address and a size in bytes. The VMM verifies that the region is contained in a single object. It is possible that a specified region will not be page-aligned. In such cases, the region may cover only a portion of the page on which it starts and only a portion of the page on which it ends. How are these partial pages to be treated? Partial pages are included in the range that is to be affected. VMM converts the byte granular size to page granular by adding the page offset of the given address to the size, and then rounding it up to the nearest page bound value. The address is also rounded down to the nearest page bound value. Basically if part of a valid page is included in the request for invalidation, that whole page is invalidated. In the same way, the whole page is committed if only part of it is requested to be made valid.

Once the set of affected pages has been determined, the VMM calls the Page Manager to do the real work.

The serialization requirements for changing the type of a region of a memory object are the same as those for reallocating a memory object.

Object Locking and Unlocking

When a region in an object is locked, the Page Manager will be called in order to load or swap in any not present pages in that region, and the thread will be blocked while the pages are made present. The locking thread must own the object semaphore in order to prevent the object from being altered (e.g., by a reallocation request) by another thread.

A successful lock request causes all the pages in the locked region to remain present and fixed at least until the lock is removed. A lock request may specify if contiguous physical memory is needed for the region. For example, contiguity is required by "dumb" DMA controllers.

The maximum size that may be locked contiguously is 64K bytes. Requests to lock regions non-contiguously will succeed so long as sufficient physical memory is available and the Page Manager's overcommitment requirements are satisfied. Requests to lock regions contiguously that meet the criteria for non-contiguous locks can be granted if any of the following statements are true:

- The region to be locked is contained in a single page.
- All the pages in the region to be locked are already contiguous.
- None of the pages in the region are already locked.

If the lock request cannot be satisfied for any reason, it may be necessary to block the requesting thread until the request can be satisfied. If the Page Manager blocks the requesting thread while it holds the object semaphore, then any other threads attempting to lock regions of that object will be blocked even if their locks would otherwise succeed. In order to prevent this unnecessary serialization from occurring, the Page Manager will return a block i.d. to the VMM who will release the object semaphore and block using the given block i.d. When the blocked thread is awakened, the VMM must restart the operation from scratch since some other thread may have run first and freed or reallocated the object.

In addition to specifying whether a lock requires contiguous physical memory or not, the requestor must specify the duration of the lock. A lock may either be "short-term" or "long-term". These types behave differently in the way that they affect other operations on the object being locked.

The algorithm for processing a lock request is as follows:

1. Round the address and size of the region to be locked to page granularity.
2. Get the information for the enclosing object from the given address and context, and take the object's semaphore if we can do so without blocking. Repeat until the object is known to be stable.
3. If not a VDM task, check address range contained in one object.
4. Check if lock counts in object record would overflow. Error if so.
5. If a device lock request, delay while there are threads waiting on short-term locks.
6. Call the Page Manager to lock the pages. If the lock request failed because of short term locks conflicting with a request for contiguous pages and we are allowed to wait, indicate that we're waiting, release the object semaphore, wait for the locks to clear and repeat the whole operation.
7. If a long-term lock request, record the arena record handle and PTDA handle pair in long-term lock database.
8. Clear the object semaphore.

If a lock request is successful, a 96-bit lock handle is returned to the caller. The Page Manager will provide an interface to return the VMOB handle given the lock handle.

A lock is removed by passing the lock handle to the VMM's unlock entry point. Ownership of the semaphore for unlock operations is never required. A thread may block during the process of removing a short-term lock or a long-term lock from an object.

The algorithm for processing an unlock request is as follows:

1. Get the object's handle from the lock handle by calling the Page Manager
2. Obtain object's semaphore. This step may block if semaphore is already owned by another thread.
3. Call the Page Manager (PGUnlock) with the lock handle to unlock the specified range of pages. If the Page Manager returns successfully, it returns an indicator of the duration of the lock so that the appropriate count in the VMOB can be decremented.
4. If the lock removed was a short-term lock, then decrement the short-term lock count. If this count goes to zero, and the lock-wait bit is set (this bit is the low-order bit of the short-term lock count), then run any threads waiting for short-term locks and clear the bit.
5. If the lock removed was a long-term lock, then decrement the long-term lock count. If this count goes to zero, and the lock-wait bit is set (this bit is the low-order bit of the long-term lock count), then run any threads waiting for long-term locks and clear the bit.
6. Release handle semaphore, wake any threads waiting for the semaphore to be freed.

Aliases

Aliases are any mapping of a given memory region into a task's private or global address space. This mapping has to remain within the boundaries of a given memory object. Mappings crossing memory objects are not permitted. Aliasing is needed by several kernel components for a variety of reasons. DevHlp services use it to make system memory accessible from a given task's address space, or vice versa. VDM Manager uses it to implement Extended Memory Specifications by aliasing low memory regions to addresses above 1Meg. DosDebug uses it to give the debugger access to debuggee's memory for the purpose of reading or setting breakpoints. Cs aliases of Selector Manager are implemented by creating a local alias for the region in the same task's address space and mapping the corresponding selector as code.

One major difference between shared memory and aliases is that the shared memory lives at the same linear address and have exactly the same size across all contexts. Aliases can be made at different addresses and usually do not map the whole memory object.

Every alias created has an VMAR linked to the corresponding memory object's VMOB This record keeps information about the context, linear address and size of the alias. Additional data is kept in alias records, managed by BMP32. For every alias, there is a corresponding alias

record that keeps information about the offset of the alias from the start of the memory object, the type of the alias (VDM, DosDebug, Devhlp etc...), and the context alias was created to (important if object is shared). This alias record is doubly linked to its arena record via alias and arena records handles. (see the figure) When the information in the alias VMAR is combined with that of in the alias record, all needed knowledge about the alias mapping is obtained.

Discardable Object Management

There are no plans to support objects discardable as a whole in current version. In OS/2 v 2.0, discarding is done on a per page basis, therefore is invisible to VMM. In order to be compatible with OS/2 v 1.x, discardable memory objects allocated runtime via DOSALLOCSEG will be allocated as swappable and never discarded. Therefore the calls via old APIs DOSLOCKSEG and DOSUNLOCKSEG will always return success on valid, user accessible data without doing anything.

Kernel Data Management

1. BMP32

BMP32 is the flat model Block Management Package. It is designed to manage pools of fixed size blocks, allowing dynamic resizing of the pool. To create a pool, a client component calls the initialization entry point specifying the size of a block, the address of a function that takes a pointer to a block and identifies it as busy or free, the maximum number of blocks in the pool, and whether the pool is swappable or fixed. BMP32 allocates a sparse object of the appropriate size (including space for management data) and type out of the system arena. A special initialization entry point exists for use by the VMAR management routines and the VMOB management routines, since they will need to set up their pools before the system arena is initialized.

For each pool, BMP32 maintains a linked list of free blocks. A call to allocate a block takes the first block off this list. When this list is empty, BMP32 will attempt to create more free blocks by committing one or more pages of storage to the end of the pool. BMP32 also maintains a pointer to the highest busy block in the pool. When this block is freed, BMP32 will use the provided busy block identifier function to find the new highest busy block. If this block is low enough that one or more entire pages at the end of the pool are free, then BMP will walk the free block list to remove any blocks that reside on that page, and then it will free the page. BMP32 will use the first four bytes of each free block as its free list link. Thus, clients must place whatever information is used to distinguish busy blocks from free ones after the first four bytes. BMP32 will allocate zero-initialized pages when it grows a pool, so a bit or byte equal to zero should be used to indicate a free block. Once a block has been allocated for the first time, it is up to the client to zero the free indicator before calling BMP32 to put the block back on the free list.

For a block size of exactly four bytes, there is no room in the block for the free indicator. Such clients have two choices. They can provide a null pointer instead of a free block identifier function, in which case, BMP32 will assume the block preceding the freed block is the new highest busy block. Obviously, if this block is actually free, then the pool may not be shrunk every time it is possible to do so. The other choice is for such clients to maintain their own auxiliary data structure, such as a bitmap.

BMP32 is expressly designed so that pools may be treated as tables. The BMP32 initialization routine will return the address of the first block in the pool. BMP32 will export an interface (probably a macro) that, given the address of the first block, returns the address of the highest busy block. Clients can use this address to check that they do not try to make references beyond the current end of the pool. Such references may cause fatal page faults.

2. RMP

RMP is the Record Management Package. It is useful for managing variable length records that are not required to reside at fixed virtual addresses. It makes efficient use of space by compacting the records when the amount of free space in the object being managed passes a certain threshold. A record is found by linear search of the user-defined key value within each record.

Because of compaction, clients should not expect pointers returned by lookup operations to be valid after blocking or yielding the CPU. The current plan is to support the existing 16:16 interface only. If a 0:32 interface is required, it will be built on top of the kernel heap facilities.

3. Kernel Heaps

Kernel heaps are intended for use by those components of the kernel that require variable length blocks of system memory with granularity finer than one page (4K bytes). Such clients are not good candidates for either BMP32 (fixed length blocks) or RMP (compaction changes virtual addresses). By using a kernel heap, one does not incur the overhead of regular objects, namely 4K granularity and a significant amount additional system data structures. Objects requiring handles, like MTEs, can be allocated out of a kernel heap and given pseudo-handles. Kernel heap blocks are assigned 4-byte granular addresses and sizes. The virtual address range occupied by a kernel heap block remains constant except possibly on reallocation operations. Block growth will require a change to a new address range entirely if a succeeding free block cannot be utilized. Block shrinkage does not alter the base address of the block.

- Generic heap allocations

For the allocation of generic writable data, two general-purpose kernel heaps are provided, one for blocks that must remain resident in memory and one for blocks that may be swappable. Special interfaces are provided for allocation, reallocation and freeing of blocks from these heaps. These interfaces provide a subset of more general heap interfaces, and comprise the following functions; for the resident heap `allocKRHB()`, `reallocKRHB()` and `freeKRHB()`; for the swappable heap `allocKSHB()`, `reallocKSHB()` and `freeKSHB()`.

- Special heap allocations

In addition to the allocation of generic writable heap blocks, a more general heap interface is provided. Function `VMCreateKH` creates a new heap; use of this function is intended to be limited primarily to system initialization since each heap incurs a per-heap memory overhead. Functions `VMAallocKHB()`, `VMReallocKHB()`, `VMFreeKHB()`, `VMShrinkKHB()`, `VMMoveKHB()`, `VMGetInfoKHB()` perform operations on blocks of a specific heap. Heap blocks may be allocated with the special properties listed below; some such as swappability apply to all blocks on a heap, others, such as contiguity are individually specifiable.

- Swappable or resident memory (specifiable at heap creation time only)
- Read-only or writable memory (specifiable at heap creation time only)
- Physical memory with addresses < 1Meg (specifiable at heap creation time only)
- Contiguous physical memory (resident blocks only).
- Physical memory addresses <= 16 Meg
- Alignment to a page boundary or paragraph (16 byte) boundary
- Swappable blocks <= 4K not crossing a page boundary
- User addressible memory (for sysinit and device driver init at ring 3)

- Selector-assigned heap allocations

Heap blocks can also be allocated (etc.) with GDT selectors mapping them. The Selector manager provides functions `SELAllocKHB()`, `SELReallocKHB()`, `SELFreeKHB` and `SELMoveKHB()` for this purpose.

- Basic resident heap structure

A resident heap is implemented as a single sparse memory object. The memory object has a header (for writeable resident heaps, the header is placed at the beginning of the object, for read-only resident heaps, the header is contained in a heap block allocated from the writeable public heap. The figure shows the structure of a resident heap header. At any given point in time, the entire virtual address space of a heap is divided into allocated and free blocks. Each block contains a 4-byte header preceding the data. Large blocks (of size exceeding 7ffc hex bytes including the header) and selector-mapped blocks contain in addition, a 8-byte data area at the end of the block called the "attribute area". The figure shows the format of a regular heap block header, The figures show the format of an attributed heap block header and the format of an attributed heap block's attribute area.

Blocks on a resident heap are linked together via a singly linked list; traversal from one block to the next is done by adding the size of the block (held in the header or attribute area) to its header address.

In addition to the singly linked list, free blocks are linked via a doubly linked list, using the first 8 bytes of the data (i.e. immediately following the header) for the links. This list is circular; a dummy free block embedded in the heap header serves as the anchor block. Initially, the free list contains one gigantic free block. The free list is not sorted completely by size; instead blocks are divided into several hierarchic size classes. Each class has maximum and minimum block size limits. All blocks in the class have size (including header and attribute area) within the limits. The minimum size limit of a class is 1 + maximum size limit of the next lower class. To speed up allocation of blocks, the free list is arranged so that the blocks in each class are adjacent on the free list. All blocks of the lowest class appear first on the free list, followed by all blocks in the next higher class and so forth. On an allocation request, the lowest class to which a free block of the smallest suitable size would belong is determined and only the portion of the free list beginning with that class is examined. This strategy means that the need to search through a lot of small blocks to find a larger block is minimized. This algorithm is implemented by maintaining a set of "sideways" pointers into the free list as an array in the heap header. Each pointer points to the first block of a given class on the free list; if a class is exhausted, the pointer points to the first block of the first non-empty class above the exhausted class.

Coalescing of free heap blocks is performed when a block is freed, or when a coalescing restriction is lifted on return from a yield at the block during a search. Coalescing involves merging with at most both of the immediate neighbors of a block. Finding the address of the succeeding block is straightforward, since the size of the block being freed is simply added to the block's header address. To find the address of the preceding free block (if any), a bit in the header of the block being freed indicates if the preceding block is free. If this is the case, a back pointer to that block is normally held in the last 4 bytes of free block's data (i.e. preceding any attribute area). In the special case of a free block of minimum size (hex c bytes including header), no back pointer is stored, since the last 4 bytes of the block are needed for the free list back link. To get around this problem, bit 1 of the location that would normally contain the back pointer is set if a back pointer to the block is stored and clear if a free list back pointer is stored. In the latter case, the size of the preceding free block is known to be C hex and so no back pointer is needed as the block address is just the address of the block being freed minus c hex.

- Basic heap block allocation algorithm

A request to allocate a heap block is validated to determine if blocks on the free list are at all eligible; requests such as a multi-page contiguous block cannot be satisfied from the existing free list. If the free list is usable, the search algorithm selects the minimum size category applicable to the request size and linearly searches the free list from that point on for a first fit. Blocks belonging to one size category are immediately followed by blocks belonging to the next higher category; the effect of this is that a request will be automatically fulfilled by a block of a higher category when a lower category is empty. During a free list search, yields occur at intervals in order to minimize dispatch latency for other runnable threads. If no blocks on the free list can be used to fulfil the request, heap growth is attempted by committing sufficient pages of backing storage at the end of the existing committed region.

Concurrent heap block transactions are supported and serialized where necessary by the use of one semaphore per heap. The semaphore does not have to be held during a free list search; instead, a search is made and then an attempt is made to take the semaphore without having to yield, and if the semaphore is busy, a new search will be undertaken after a wait and the operation will be repeated until interrupted, the semaphore is taken without yielding or the free list has no suitable blocks.

- Basic heap block free algorithm

When a heap block is freed, it is automatically coalesced with its two immediate neighbors if they are free and no yield has occurred at them during a free list search. Location of the preceding block (if free) is made possible by a back pointer to its header kept in the end of that block. It is valid to interpret this location as a back pointer only if a special flag, the "preceding block is free" flag is set in the current block's header and the preceding block's size exceeds 12 bytes. The newly freed block, coalesced if possible is inserted into the free list at the head of its block category. An array of pointers into each block category is maintained; those pointers referencing the old head are updated.

- Basic swappable heap structure

The swappable heap is currently implemented as a singly linked list of busy and free blocks with headers allocated from a resident BMP32 object. Block allocation currently involves a linear search beginning at the first free block (if any). Re-implementation modeled on the resident heap design is anticipated. To avoid having to page-fault in many swappable heap pages to do a search, swappable heap block descriptors held in blocks of a resident BMP32 object replace the block headers that the resident heap uses. This adaptation is used in the existing implementation of the swappable heap. Each swappable heap block is prepended with a back pointer to the corresponding block descriptor.

Interfaces

1. API

All existing API memory management calls will be fully supported. In addition, new API calls will have to be added in support of memory addressing with 0:32 format.

- VMAlloc/private

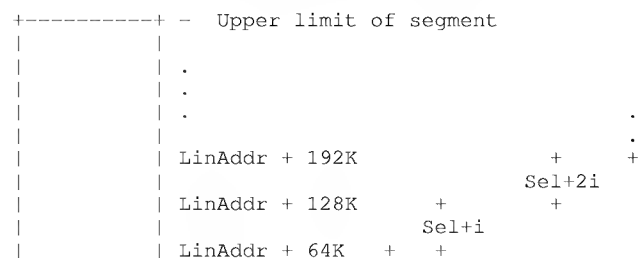
Allocates private memory. The memory object allocated with this API call can be any size from 1 page to 4G. In the first release, it will not be possible to satisfy a request larger than 512Meg.

All memory will be allocated with page granularity. Any size request that is not a multiple of page size will be rounded up to nearest page size during allocation. The linear address of the start of the memory region will be returned upon successful allocation.

If requested, the memory object will be aliased with selectors that are HUGEINCR apart, each mapping a consecutive 64K of memory region. (see the figure). The selector value can be obtained from the returned linear address by:

$$\text{Selector} = ((\text{highword of linear address}) \ll 3) \mid 4 \mid \text{CPL}$$

This value obtained (say Sel) will map the first 64Kb of the memory object. Sel + HUGEINCR will map the region from 64K to 128K. Sel + 2*HUGEINCR will map the region from 128K to 192K, and so on. This mapping will be permanent.



```

      |-----|
      +-----+ LinAddr      Sel
                                +

```

Data segment and 64K subsegment map.

Memory objects can also be allocated as sparse. Memory allocated as sparse will have virtual memory (and selectors if requested) allocated without corresponding physical memory. There will be no memory commitment for sparse objects, and all the selectors, although reserved, will be invalid. All pages of a sparse object will be invalid after allocation, and attempt to access any invalid page within this memory range will cause termination of the process with a GP fault. The pages within the sparse memory should be validated (physically allocated) via call to memory region before being accessed. This allows applications to allocate very large virtual memory, larger than possible if the memory commitment were done during allocation. It also makes it possible for applications to manage their memory in terms of pages, have discontinuous valid and invalid pieces in their memory region. The descriptors will be mapping only up to the highest committed address in the object.

In OS/2 v 2.0 all memory objects are going to be tiled, whether tiling is requested or not. However, this will not be true for succeeding versions of os/2. Any application that relies on automatic tiling will be broken in succeeding versions when objects that are allocated without the selector tiling flag are moved above 512Meg and allocated without selectors.

- VMAlloc/shared

Allocates named or unnamed shared memory. The same features available through VMAlloc/private are also available through VMAlloc/shared.

- memory region

This API call will validate, invalidate, or answer queries about the state of physical memory mapped by a memory region. Attempt to access invalid memory addresses will cause process termination. Request to validate pages will cause physical memory allocation. Any memory region in the process' linear address space can be queried, or validated via this call, but the corresponding linear memory must be preallocated with any memory allocation APIs.

This system call will work with readable and writable, ring 2 or 3, private or shared segments created by old APIs and memory objects created by new APIs. Request to invalidate regions in shared segments/memory objects will be denied.

Validation and freeing of memory regions will be done with a page granularity. All pages that are fully or partially covered by the memory range will be affected.

- VMFree

This API will free the memory objects allocated via VMAlloc/private. It only works with memory objects as a whole, frees all virtual memory as well as any valid physical memory that may be mapped to it.

Besides these externally visible changes, there will be some internal design changes in VMM, mandated by the existence of page based memory management, and 32 bit Linear Memory Address Space.

- All memory objects will have to start at a 64K boundary in Linear Address Space (Offset zero in a segment must correspond to the first byte of a memory object, and low 16 bits of a linear address corresponds to the offset in segmented address. Therefore the low sixteen bits must be zero for the starting address of a memory object.).
- SegLock/SegUnlock interface will be expanded to support piecewise locking of memory objects. Kernel will be able to specify which offset in the memory object the lock is to start from and the size of the lock. Additional flags will indicate if physical contiguity of locked pages is required, if this lock is for read or write request (so the Page Manager will know when to set dirty bits). The lock handle returned by SegLock will be 32 bits wide. The lock handle will be set up and implemented by Page Manager.

2. DevHlp

Several new DevHlp functions will be introduced to enable device drivers access memory in flat model. The value of flat selector, one that maps the whole linear address space will be exported to device drivers during initialization. This selector should be loaded to DS : ES every time any of the new functions are called. Following are the new DevHlp functions supported by VMM

- dh_VMLock:

This function locks into physical memory a range of linear addresses. Its input parameters are linear address and size of the region to be locked, action flags, and a location to write the physical address of the locked region. Caller to this routine can specify the duration and type of lock via action flags. If the memory to be locked is not available, the caller can specify whether to return failure immediately or block until pages are available. Another flag specifies if the lock is for a short term or long.

The write bit in action flags must be set if the caller plans to write to the locked region via DMA. This will cause the dirty bit in page table entries to be set. DMA does not set this bit automatically and the data written via DMA may be lost if this flag is not specified. Another flag will specify whether the memory ought to be locked at a physical address below 16Meg. This is to support devices that cannot access physical memory above 16Meg.

The contiguity flag will cause the locked memory to be contiguous in physical memory. If this flag is set, the range of lock is limited to 64Kbytes. This kind of lock may cause byte by byte copy of physical pages in memory, therefore will be considerably slower than those that do not require continuity. In order to avoid major performance impact, device drivers should not request continuity whenever possible. Physical addresses within the same page will always be contiguous. Device drivers can avoid the contiguity requirement by keeping their data transfer within the same page and updating physical address each time they cross a page boundary.

If the lock is successful, a 96 bit lock handle is returned. This function will check for accessibility to the memory region before locking it, therefore it is not necessary to call another DevHlp function to verify this.

- `dh_VMUnlock:`

This function will unlock the region of memory locked via `dh_VMLock`. It will take 96 bit lock handle returned by `dh_VMLock` as its input parameter.

- `dh_VMAlloc:`

This function allocates virtual memory, and depending on its action flags it either commits physical storage or maps virtual memory to given physical address. The linear address of the allocated memory is returned upon success. Virtual memory is allocated in global address space unless private process space is requested. The memory allocated can be specified as swappable, resident, fixed, or contiguous fixed. If requested, fixed memory can also be allocated below 16Meg physical address. Process private memory can not be allocated as fixed, contiguous or resident.

This process memory can also be registered under screen group switch control. In this case, write access to this memory will be denied unless the process is running in the foreground.

If physical memory map is requested, the physical range must be fixed or locked.

- `dh_VMFree:`

This function frees the fixed memory allocated via `dh_VMAlloc` or unmaps the mapping created by `dh_ProcessToGlobal` or `dh_GlobalToProcess`.

- `dh_VMGlobalToProcess:`

`dh_VMGlobalToProcess` creates a mapping to global address range in the user address range. The address range can not cross memory objects. If selector map is requested, selectors will be allocated as ring 3 data. Existing hardware may not support read only mapping. Applications can run in Ring 2 privilege level and this overrides write protection page faults on the Intel 386 chip.

- `dh_VMProcessToGlobal:`

`dh_VMProcessToGlobal` creates a mapping in global address range to the user address range. The address range can not cross memory objects. Existing hardware may not support read only mapping.

The VMM's main tasks are to create and destroy private arenas, to manage the allocation, reallocation, and freeing of objects within arenas, to maintain information about which processes have access to which objects, and to maintain information about aliases within objects. In order to perform these tasks, the VMM manipulates the following data structures:

- **VMAH**

An VMAH is used to maintain information about the location of the VMM data structures associated with a particular arena, namely the VMAR list and the bitmap and hash table used to perform arena record lookup. The figure shows the format of an VMAH.

VMAH

- **VMAR**

An VMAR is used to maintain information about the allocated regions, or contents, of an arena, and the context. The figure shows the format of an VMAR.

VMAR

An VMAR contains the base address and size of an object as well as links to the next and previous VMARs in the arena. Since objects are always page-aligned, the low-order 12 bits of the address field are used as flags. An arena is represented by a doubly linked list of VMARs sorted in ascending order of base address. Objects may be contiguous, but they are not allowed to overlap. An VMAR also contains the handle of an VMOB which contains additional information about the object. There is a field in the VMAR to indicate the process which arena contains the VMAR. In some cases (e.g., sharing of code among multiple instances of a particular program), a private arena entry may represent a shared object. In such a case, VMARs in different private arenas would contain the handle of the same object's VMOB. In order to facilitate the enumeration of all contexts in which such an object

is shared, the VMAR contains a link field so that the VMARs in different arenas can be chained together. The VMOB points to the first VMAR in the chain. When an object in the shared arena is shared among multiple processes, then this link field points to a Context List. Another field in the VMAR points to the alias record if the VMAR is representing an alias.

- **VMOB**

It is a reasonably accurate generalization to say that an object represents a commitment to provide physical storage. The logical linkage between an object and its storage are provided by its linear address, its size, and its contexts. Given this information for any object, the Page Manager can find all of its data structures that describe the storage for that object, be it physical memory or swap space or a load image. This information can always be determined starting from an object's VMOB. The figure shows the format of an VMOB.

VMOB

An VMOB always points to at least one VMAR. If the object is a shared object in a private arena, then this VMAR may point to additional VMARs in other private arenas. The classic example of this case is the code object for CMD.EXE which will be shared among all instances of the program. An VMOB may point to other VMOBs. The classic example is an instance data object in a DLL. Every time a new client process links to the DLL, a new object must be created for each instance data object in the DLL. These objects all share the same region of the shared arena, but each has its own physical storage. Thus, multiple VMOBs point to the same VMAR. These VMOBs are linked together by handle in a chain, and the VMAR points to the first VMOB in the chain. In this case, the VMOBs, not the VMAR, contain the context information. In some cases, multiple processes share the same object in the shared arena, for example, DLL shared data. In this case, there is one VMAR and one VMOB, but context information is still needed, since the Page Manager maintains separate page tables for each process. For such objects, the VMAR will contain a link to a Context List which contains the identities of all the processes mapping the object into their address space.

- **VMCO**

A Context List is a linked list of VMCOs. Each record contains a link field and a context identifier. New VMCOs are chained onto the list as needed. The figure shows the format of a VMCO.

VMCO

- **Aliases**

The alias records are allocated out of BMP. Every alias will have its own VMAR as well as an alias record. Alias records have VMAR handles stored in them, and VMARs keep the handle to corresponding alias records. This way, a simple mapping function is required to go from an alias record to VMAR and back. An alias record contains information about the offset of the alias from the start of memory object, flags indicating type of the alias, and the context the alias is created from. (see the figure) All aliases to memory objects will start and end at page boundaries. The low 12 bits in the pageoffset field in local alias structure are available for flags. CS and huge aliases will also be maintained in the same bmp segment. The offset field in cs alias structure is replaced by dataset field giving the value of the data selector, and refcount field keeping reference count and cs flags. The value of the code selector will be stored in the owner field.

Alias Records Structure

- **BMP Header**

The figure shows the format of a BMP pool header.

BMP Header

The header contains a pointer to the first free block in the list of free blocks, a pointer to the first byte past the last existing block, a pointer to the first byte past the last block for which virtual space has been reserved, a pointer to the last busy block, a pointer to the free block identifier function, the size of a block, and the handle of the pool memory object.

- **Resident heap header structure**

Each resident heap has a header. For writeable heaps, the header is placed at the beginning of the heap object; for read-only heaps, the header is placed in a heap block allocated from a writeable heap. The figure shows the format of a resident heap header.

VMKRH - Kernel Resident Heap Header

Resident heap header format.

- Resident heap block structure

Each resident heap block consists of a header followed immediately by the block's data followed in some cases by the "attribute area". Headers are not visible to clients; an allocation request returns a pointer to the block's data. To minimize storage overhead, non-selector mapped resident heap blocks having total size (including header) not exceeding 7ffc hex (32764) have a 4-byte header. The figure shows the format of a regular resident heap block header.

Bits	Name	Description
0	block type flag	0 for regular block
1	yielded flag	1 if search yielded at block
2-14	size	Size of block (dwords)
15	prev free flag	1 if previous block free
16-31	block owner	Owner of this heap block

Resident heap regular block header format.

The owner id is either a special kernel owner, one of which is designated as the free block owner, or a memory handle or pseudo-handle. Currently, the only owners other than special kernel owners are MTE pseudo-handles for heap blocks containing data or code of an MTE object. The "prev free" flag indicates that the preceding block is free; note that some thread may have yielded at that block. The size is in double words (meaning 4-byte long words) and includes one double word for the 4-byte header. All heap blocks are a multiple of 4 bytes in size. The "yielded" flag indicates that a search of free blocks has paused at that point in order to relinquish the CPU. Such a block cannot be coalesced because the yielding thread has a pointer to the block's header.

Resident heap blocks having total size (including header) exceeding 7ffc hex, or selectors mapped to them have a different header format. These blocks are called "attributed blocks" because in addition to the header at the beginning of the block there is an 8-byte "attribute area" at the end of the block. By making all blocks have the same size header at the beginning of the block, it is much easier to allocate blocks whose client data starts on a given address boundary (e.g. page or paragraph). Further, the attribute area is a structure which can readily be expanded without major code change should there be a need to record additional kinds of per-heap-block information (for example locks). The figure shows the format of an attributed resident heap block header.

Bits	Name	Description
0	block type flag	1 for attributed block
1	yielded flag	1 if search yielded at this block
2-24	size	Size of block (dwords)
25-29	Unused	Reserved for future flags
30	prev free flag	1 if previous block free
31	Unused	Reserved

Resident heap attributed block header format.

As mentioned previously, attributed blocks contain an additional data area called the attribute area at the end of the block. The attribute area has the following structure.

Resident heap block attribute area format.

Page Manager Imports

PGAlloc

Performs overcommit accounting, allocates page tables and storage, initializes virtual to physical mappings. Callers VMAllocMem

PGAttach

PGFree

Performs overcommit accounting, frees page tables and storage, destroys virtual to physical mappings. Callers VMFreeMem

PGGetType

PGLock

This function locks down a range of pages in the given context so that they will be present and may not be paged out until they are unlocked. In addition, it is possible to specify that the given pages must be locked contiguously and at physical addresses below 16 megabytes. Callers VMLock

PGLockCPage

Given a lock handle, this function returns the number of pages in the locked region. Callers VMUnlock

PGLockHOB

Given a lock handle, this function returns the handle of the object record for the object in which the locked region starts. Callers VMUnlock

PGLockLong

Given a lock handle, this function returns true if the lock is long-term and false if it is short-term. Callers VMUnlock

PGPageToPhys

PGSetState

PGSetType

PGUnlock

Given a lock handle, this function unlocks the specified region. Callers VMUnlock

Design Constraints

To be added.

Design Review

To be performed.

VMM Functions

Exported functions

VMInit - Initialize the Virtual Memory Manager
VMExchangeSel - Move selector from one object to other
VMInitProtect - Initialize protected region of shared arena
VMReserve - allocate virtual memory
VMAllocMem - Reserve virtual memory and commit initial physical storage
VMReserveMem - reserve virtual memory
VMMMapVDM - create/destroy mapping in VDM address space
VMSelAlias - Export address of Selector Manager local alias record.
VMMMapAlias - Map an alias to memory region
VMMMapDebugAlias - Map a local alias to memory region for ptrace.
VMGetAliasOffset - Return the offset of the given alias from the object
VR32AliasMem - Create an alias to a memory object.
VR32AllocMem - Allocate a memory object in the private arena.
VR32AllocSharedMem - Allocate a memory object in the shared arena.
VR32AllocProtectedMem - Allocate a protected memory object
VR32GetNamedSharedMem - Attach to existing named shared memory object.
VR32GetSharedMem - Attach to existing shared memory object.
VR32GiveSharedMem - attach a target process to existing memory object.
VR32FreeMem - Free a memory object.

VR32SetMem - Commit/decommit memory, and change access permissions.
VR32QueryMem - Get page information.
VR32QueryMemState - Get page state information.
VM32ApiAliasMem - Create an alias to a memory object.
VM32ApiAllocSharedMem - Allocate a memory object in the shared arena.
VM32ApiGetNamedSharedMem - Attach to existing named shared memory obj.
VM32ApiQueryMem - Get page information.
VM32ApiQueryMemState - Get page state information.
VR32IAllocThreadLocalMemory - Allocate thread local memory
VR32IFreeThreadLocalMemory - Free thread local memory
VR32ISYSCTL - General purpose PRIVATE system interface
VMGetOd - Return Memory Object Virtual Memory Manager Data
VMOiTOOd - Generate object data from object info
VMOdTOOi - Generate object info from object data
VMInitTask - Create/Initialize Per-task Memory Management Data
VMFreeTask - Free Task's Memory Management Data Structures
BMPDestroy - Destroy a BMP object.
BMPFree - free a block
BMPGet - get a free block
BMPGet2 - get a free block with or without blocking
BMPHandleMap - Verify and map a BMP 32 handle to a record pointer
BMPInit - initialize BMP object
dhw_VMLock - Worker function for DevHlp interface dh_VMLock.
dhw_VMUnlock - Worker function for DevHlp interface dh_VMUnlock.
dhw_VMGlobalToProcess - Worker function for DevHlp interface dh_VMGlobalToProcess.
dhw_VMPProcessToGlobal - Worker function for DevHlp function.
dhw_VMAAlloc - Worker function for DevHlp interface dh_VMAAlloc
dhw_VMSetMem - Worker function for DevHlp interface dh_VMSetMem
dhw_VMFree - Worker function for DevHlp interface dh_VMFree
VMLinToPhys - Convert linear address to physical
dhw_VMInitAlloc - Worker function for DevHlp interface dh_VMAAlloc used during initialization.
VMDH_AllocRemoveAccess - Remove user level access from the objects
VMGetLaddr - Return the linear address of a memory object given its handle.
VMTrap2FindArena - given hob : find starting linear address of arena and the pptda that it resides in.
VMObjHandleInfo - Find context and address of the given object handle.
VMQueryArenaAvailMem - Query available memory in given arena.
VMQueryMem - Query page granular permissions and attributes
VMIsPrivatized - Return true if given memory is private/privatized
VMCopy - Copy specified number of bytes from one memory location to another.
VMLockMem - Lock a region of an object
vmLockHandleSig - Lock Handle Signature
VMUnlock - Unlock a Region of an Object
w_AllocShrProtSeg - Create a Protected Named Memory Segment, 0 - 64K

w_AllocShrSeg implements the DOSALLOCSHRSEG system function, issued by tasks to create a shared memory area with the specified name and size. **w_AllocShrProtSeg** allocates the memory segment from protected memory.

```

ENTRY    w_AllocShrSeg:
    BX = size of shared area, in bytes (0 means 65536)
    DS:DX = address of name string - must start with \SHAREMEM\
EXIT     'C' set if error
        AX = ERROR_INVALID_NAME
        AX = ERROR_ALREADY_EXISTS
        AX = ERROR_NOT_ENOUGH_MEMORY
        AX = ERROR_INTERRUPT
    'C' clear if OK
        AX = selector to shared memory segment
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EFlags
ENTRY    w_GetShrSeg:
    DS:DX = name string
EXIT     'C' set if error
        AX = ERROR_INVALID_NAME    (if badly formed name)
        AX = NAME_NOT_FOUND
        AX = ERROR_NOT_ENOUGH_MEMORY
        AX = ERROR_ACCESS_DENIED
    'C' clear if OK
        AX = selector to shared memory segment
USES     EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EFlags
INPUT:
    AX      Current (Local) Selector
    DS:DX   Address of Name String
EXIT-NORMAL:
    AX = New (Global) Selector
    CLC
EXIT-ERROR:
    AX != 0
        Invalid Name
        Name Exists
        Invalid Selector
  
```

STC

EFFECTS:

```
    Convert given segment to Read-Only, Global,
    Shared Data Segment
Procedure w_GLOBALSEG
check name syntax via CanonBuf
obtain shared memory semaphore via GetShrSem
if (no error) then
    validate name via ValidateName
    if (no error) then
        insure name does not exist via Rmp_Find
        if (no error) then
            set error
        else
            add shared record via Rmp_Add
            if (no error) then
                obtain info from local via GetDescInfo
                if (no error) then
                    Call _SELAllocKHB to reserve memory and GDT selector
                    Call _VMCopy to copy over the segment
                    (Read Only, Shared, Data)
                    Update handle : selector in shared memory record
                    VMFreeMem(free the local LDT image of the segment)
                else
                    set error code = ERROR_ACCESS_DENIED
                    Free shared rmp record via Rmp_Free
                endif
            endif
        endif
    endif
    release shared memory semaphore via ClearShrSem
endif
free temporary storage via FreeTempBuffer
return
End Procedure w_GLOBALSEG
```

VMHandleVerify - verify that handle is valid
VMHandleMap - map an object handle to its object record
VMGetSem - Get ownership of Object semaphore
VMGetPTDAAddr - Return the virtual address of a PTDA
VMGetHandle - find object record handle
VMQueryMaxAlloc - return the size of the largest allocatable object
VMSGSControl - Add the address range to SGS list
VMSGSVAlidate - Walk SGS Control list to (in)validate page tables.
VMGetOwner - Get owner of selector-mapped memory object.
VMClearSem - Release a Memory Object's semaphore
VMCreatePseudoHandle - create a pseudo-handle and set mapping
VMSearchRef - Search references for a given shared memory object
VMIsAttached - See if given object is attached to a process.
VMEachMap - Apply the worker function to some or all maps to a given memory region.
VMScanMaps - Call worker function for every mapping of a single page.
VMAttach - Attach a given shared memory object to a process.

_VMGetTempBuffer obtains a temporary buffer from the Temporary Buffers BMP32 object. If there are no free records left, and we can't grow the temporary buffers object, this routine will block until a buffer is freed.

WARNING! This code depends on the fact that BMP32 does not do sparse decommit, i.e. all pages of a BMP32 object remain committed while there is at least one busy BMP32 block beyond them.

WARNING! There is a new hVPB field at the beginning of the temp buffer for the MFT search code so that the hVPB and the path can be one contiguous bit string to be entered into a tree data structure (Patricia Tree). However, most callers just want the name portion of the buffer so this routine will return the pointer to the name portion of the buffer. In order to access the hVPB part, the caller must use a negative offset.

```
ENTRY    vgt_ppBufFlat - pointer to flat address of temp buf
          vgt_ppBufSelOff - pointer to 16:16 address of temp buf or NULL
RETURN   (EAX) = NO_ERROR or ERROR_INTERRUPT
          vgt_pBufFlat and vgt_pBufSelOff filled
          in to point to temp buffer (name part)
USES     EAX, ECX, EDX, EFlags
```

_VMFreeTempBuffer frees temporary buffer allocated via _VMGetTempBuffer, and wakes up any threads blocked waiting for a Temporary Buffer.

WARNING! This code depends on the fact that BMP32 does not do sparse decommit, i.e. all pages of a BMP32 object remain committed while there is at least one busy BMP32 block beyond them.

This routine, instead of `_BMPFree` should be called when freeing temporary buffers, otherwise we may leave some threads blocked.
WARNING - There is a new `hVPB` field at the beginning of the temp buffer for the MFT search code so that the `hVPB` and the path can be one contiguous bit string to be entered into a tree data structure (Patricia Tree). However, most callers use just the name portion of the buffer so this routine takes the pointer to the name portion of the buffer.

```
ENTRY    vft_pBuf - Address of buffer to free (name part)
EXIT     NONE
USES     EAX, ECX, EDX, EFlags
```

Local Function Prototypes:

vmGetArenaHeader - return a pointer to an arena header
vmArMap - map an arena record handle to its arena record
vmArlsSentinel - returns true if the specified record is a sentinel
vmIsBusyAr - Returns true if the specified arena record is busy
vmSentAr - returns true if the specified arena record is a sentinel
vmOrdSentAr - returns true if the specified arena record is an ordinary sentinel
vmBdrySentAr - returns true if the specified arena record is a boundary sentinel
vmRegAr - returns true if the specified arena record is regular
vmArVa - return the address from an arena record
vmArPg - return the page address from an arena record
vmParToHar - return the handle for an arena record
vmHandleMap - map an object record handle to its object record
vmBMPHandleMap - map a BMP32 record handle to its record address
vmSystemOwner - Check that a handle is one of the special system owners
vmAlign - extract alignment type from flags
vmCeil - round up to next boundary
vmFloor - round down to nearest boundary
vmParaCeil - round up to next paragraph boundary
vmParaFloor - round down to nearest paragraph boundary
vmPageCeil - round up to next page boundary
vmPageFloor - round down to nearest page boundary
vmSelCeil - round up to next selector boundary
vmSelFloor - round down to nearest selector boundary
vmDirCeil - round up to next page directory boundary
vmDirFloor - round down to nearest page directory boundary
vmArAlloc - allocate an arena record
vmArFree - free an arena record
vmCoAlloc - allocate a context record
vmCoFree - free a context record
vmCoMap - map a context record handle to its context record
vmIsBusyCo - Returns true if the specified context record is busy
vmPcoToHco - return the handle for a context record
vmObAlloc - allocate an object record
vmObFree - free an object record
vmTile - return true if request is for tiled memory
vmOvflw - return true if addition will overflow
vmVDMTask - test if initialized VDM
vmCurrTask - check if PTDA is for current task
vmPrivCo - check if context record is privatized
vmPrivAr - check if given context is privatized
vmAliasAr - check if arena record is an alias
vmReloadAr - check if arena record for reload or pre-reserve memory
vmRangeWithinAr - test if given virtual address range within object
vmRangeWithinArPg - test if given virtual page range is within object
vmIsBusyOb - Returns true if the specified object record is busy
vmShrOb - test if object is shared
vmPrvOb - test if object is private
vmShrinkOb - test if object is shrinkable
vmHasAliasOb - test if object has aliases
vmResidentReq - Check if allocation request for resident memory
vmSwappableReq - Check if allocation request for swappable memory
vmSysArenaReq - Check if allocation request for system arena
vmShrArenaReq - Check if allocation request for shared arena
vmPrvArenaReq - Check if allocation request for private arena
vmShrDataReq - Check if allocation request for shareable data
vmLocAnyReq - Check if allocation request for any address range
vmLocSpecReq - Check if allocation request for specific addresses
vmLocAboveReq - Check if allocation request for addresses above
vmAlignParaReq - Check if allocation request for paragraph alignment
vmAlignPgReq - Check if allocation request for page alignment
vmAlignSelReq - Check if allocation request for tiled selector alignment (64K alignment)
vmAlignDirReq - Check if allocation request for page directory alignment

vmVDMReq - Check if allocation request for VDM
vmReloadReq - Check if allocation request for reload/load out of reserved memory.
vmRetainReq - Check if allocation request specifies retention for future reloads.
vmPreResReq - Check if request to pre-reserve virtual memory for future allocation to object(s).
vmWithinAr - Check if arena record maps given address
vmWithinArPg - Check if arena record maps given virtual page
vmIsLoadOb - check if object is a loader-created object
vmHalToPal - Convert handle of alias record to its address
vmPalToHal - Convert alias record pointer to handle
vmIsSELAlias - Return nonzero if given alias record is SELMGR alias record
vmIsDBGPrivAlias - Return nonzero if given alias record is privatized debug alias record
vmSemAssertOb - assert that we have an object semaphore
vmCrossesPgbdry - test whether address range crosses page boundary
vmGetCo - Return hco, and pco matching the context
vmGetOb - Find object record corresponding to Ar and hobptda
vmError - Call error function for debugging
vmAssert - cause internal error if assertion fails
vmInitAlloc - Allocate an object from the system arena
vmInitBMPInit - Create BMP Object within VMInit
vmFreeAlias - Free alias record belonging to given arena record.
vmCheckAlias - Check the alias record to make sure it is of desired type.
vmIsPrivLoadObj - A loader object being privatized for a debug alias
vmIsBusyAliasRec - Check if the given local alias record is busy
vmInitAliasRec - Initialize alias bmp records object.
vmAliasToAlias - Called by VMMapAlias and VMMapDebugAlias when an alias is being aliased.
vmFreeObjectAliases - Free all aliases to a given object created from a certain context.
vmChangeAlias - Change mapping of local alias to the privatized one if contexts match.
vmHasDiscardablePages - return TRUE if any of the pages in the given range is discardable
vmHasDebugAlias - return true if given range has debug aliases
vmLinkOb - Link given object record to the arena record
vmReserveSub - allocate virtual memory only
vmRecalcShrBound - Recompute lower shared arena bound
vmFreeSearchUp - search upwards for free region
vmFreeSearchDown - Search downwards for free region
vmMoveSentinelUp - try to find free space by moving sentinel up
vmMoveSentinelDown - try to find free space by moving sentinel down
vmAllocTLBAdjust - Adjust size of reservation up or down
vmfTranslate - Translate API flags to internal VM/PG flags.
vmScanContext - Call worker function for each context // [153480](#)
vmFreeMemContext - Free a memory object from a single user context
VMFreeAllContext - Free a memory object from all user context
bmpUpdateLastBusy - update the last busy block pointer
vmRebuildBMPFreeList - rebuild the free list
bmpShrink - shrink a BMP object
vmBmpGet - Get the next record on BMP free llist, updating highest busy block if necessary.
bmpIsBusy - dummy busy block identifier
vmBuildBMPFreeList - Build freelist in BMP object
vmInitBMPObject - Initialize BMP object header and free list
vmDh_AllocAddList - Add the given address to the temporary access list.
vmDh_AllocRemoveList - Remove given entry from temporary access list
vmQueryObject - Query object permissions and attributes
vmInitPublicHeaps - Initialize the Public Swappable and Resident Heaps
vmAllocKSHB - Allocate kernel swappable heap block
vmReallocKSHB - Realloc Kernel Swappable Heap Block
vmFreeKSHB - Free Kernel Swappable Heap Block
vmShrinkKSHB - Shrink Kernel Resident Heap Block
vmIsValidKSH - Check integrity of swappable heap
vmFindKSHB - Find Kernel Swappable Heap Block
vmSetupSpecialKSHB - setup for a "special" kshb allocation request
vmCreateKSH - Create a kernel swappable heap
vmMakeHdrRegKRHB - Compose Regular KRHB Header
vmMakeHdrAttrKRHB - Compose Attributed KRHB Header
vmIsHKH - Verify that given handle is a valid heap header handle
vmIsHKRH - Verify that given handle is a resident heap header handle
vmIsHKSH - Verify that given handle is a swappable heap header handle
vmWriteULKRHB - Set unsigned long word in kernel resident heap
vmWriteUSKRHB - Set unsigned short in kernel resident heap
vmCeilKRHB - Round up address to resident heap block granularity
vmFloorKRHB - round down address to resident heap block granularity
vmCanCoalesceKRHB - return true if block can be coalesced
vmYieldCheckKRHB - Update iteration count and yield if necessary
vmHasYieldedKRHB - return true if someone has yielded here
vmIsFreeKRHB - return true if block is free
vmIsFreeRegKRHB - Return true if regular block is free
vmIsFreeAttrKRHB - Return true if extended block is free
vmIsDummyKRHB - return true if block is the dummy block

vmOwnerKRHB - Get the owner of a Kernel Resident Heap Block
vmOwnerRegKRHB - Get the owner of a regular Kernel Resident Heap Block
vmOwnerAttrKRHB - Get Owner of an Attributed Kernel Resident Heap Block
vmCopyAttrKRHB - Copy Kernel Resident Heap Block Attribute Area
vmSizeDataKRHB - Get Size of Kernel Resident heap Block's Data
vmClrYieldKRHB - Clear Yield Bit In Kernel Resident Heap Block
vmSetFreeKRHB - Mark the given block as free
vmphobOwnerKRHB - Point at owner field in a KRHB
vmSetYieldKRHB - Set the yield bit for Kernel Resident Heap Block
vmSetHdrKRHB - Set Kernel Resident Heap Block Header
vmHdrKRHB - Get value of Kernel Resident Heap Block header
vmIsValidKRHB - Check that block is valid
vmHasValidOwnerKRHB - Check if heap block has valid owner
vmIsValidOwnerKRHB - Check if valid heap block owner
vmNeedsRegKRHB - Check if a regular or extended KRHB needed
vmNeedsAttrKRHB - Check if an attributed KRHB needed
vmIsWithinKRH - Test if block pointer is in range
vmSizeKRHB - return the size of the given block
vmSizeRegKRHB - Return the size of the given regular block
vmSizeRegKRHBD - Return the size of the given regular block's data
vmSizeAttrKRHB - Return the size of the given attributed KRHB
vmIsRegKRHB - Check for Regular Kernel Resident Heap Block
vmIsAttrKRHB - Check for Attributed Kernel Resident Heap Block
vmNextKRHB - Point to next resident heap block
vmNextRegKRHB - Point to next resident heap block after reg block
vmNextFreeKRHB - Point to next resident heap block on free list
vmPrevFreeKRHB - Point to previous resident heap block on free list
vmHasPrevFreeKRHB - Block has previous free block
vmHasPrevFreeRegKRHB - Regular block has previous free block
vmHasPrevFreeAttrKRHB - Attributed block has previous free block
vmIsEmptyKRHS - Check if free list is empty
vmHeadFreeKRHB - Get head free block from free list.
vmDummyFreeKRHB - Get pointer to dummy free block in free list.
vmBlockSizeKRHS - Get maximum block size for given free list section.
vmRoundSizeRegKRHB - Round up size to size of regular block needed
vmphbaKRHB - Get pointer to attribute data in Kernel Resident Heap Block
vmDecommitMinKRHB - Return pointer to lower bound for decommitted region within block.
vmDecommitMaxKRHB - Return pointer to upper bound for decommitted region within block of specified size.
vmPossWithinKH - Quick check to see if address possibly within KH

Local heap procedures (not implemented as macros)

vmAccessLongLockInfo - Access Long Lock Info

_vmAllocNamedShrMem allocates named shared memory. This procedure is C callable.

WARNING This procedure is in swappable code.

1. Validate the name : determine if the named segment exists.
2. Add 2 new records to the rmp segment, one global and one local.
3. Allocate a segment of memory from the system pool.
4. Initialize the handles and ref. counts in rmp records.
5. Return the selector to the segment, or the error code

```

ENTRY  cb    - Size of memory to be allocated
        cbCom- size to be committed (<= cb)
        pBuf - 16:16 address of the name string
        flsel- Selector flags (see SELAL_ in selxport.h)
        flvm - VM flags (see VMAC_ in vmexport.h, PG_ in
                pgexport.h)
        pac  - Pointer to VMAC buffer to return address and selector
        hmte - mte of calling module (for ownership tracking)
RETURN  EAX   - Error code (NO_ERROR if no error)
                If no error, pac->ac_sel and pac->ac_va filled with
                selector and address of the new memory
USES    EAX, ECX, EDX, Eflags
  
```

```
int  vmAllocNamedShrMem(cb, cbCommit, pBuf, flsel, flvm, pac)
```

_vmGetNamedShrMem obtains access to an existing named shared memory. This procedure is C callable.

WARNING This procedure is in swappable code.

1. Validate the name : determine if the named segment exists.
2. Search for local record. If found increment ref. count. If not add the local record and attach the segment.
3. Increment the global reference count.
4. Return the selector and address to the segment, or the error code

```
ENTRY    pBuf - 16:16 address of the name string
         fl  - VMAttach flags (PG_* flags in pgexport.h)
           PG_NOINCR means call originated from w_GetShrSeg
         pac - Pointer to VMAC buffer to return address and selector
           (Address not returned in GDT selector case)
RETURN   EAX - Error code (NO_ERROR if no error)
           If no error, pac->ac_sel and pac->ac_va filled with
           selector and address of the new memory
USES     EAX, ECX, EDX, Eflags
```

int vmGetNamedShrMem(pBuf, fl, pac)

ValidateName verifies that the pathname starts with the specified prefix.

WARNING This code is in swappable code.

```
ENTRY:   DS:ESI = address name string
         ES:EDI = address of prefix, count followed by string
EXIT:    'C' clear if name valid
         DS:ESI = address name string excluding prefix
         ECX = # of chars in name string
           (excluding prefix, including 00 terminator)
         'C' set if name format invalid
         AX = ERROR_INVALID_NAME (error code)
USES:    EAX, ECX, ESI, EDI, ES, Eflags
```

_vmFreeNamedSharedMem decrements the reference count in the named shared memory records, and frees them if necessary. Then it returns NO_ERROR if the last reference to the shared memory is removed from the given task. It returns ERROR_BUSY if the memory is not yet to be freed.

WARNING This procedure assumes that Shared Memory and the object handle semaphores are taken!

WARNING This procedure is in swappable code.

```
ENTRY    [ESP+4] pointer to object information buffer
         [ESP]   flags (Task termination or not)
EXIT     [EAX]   0 if memory is to be freed
           ERROR_BUSY if there are outstanding references
USES     EAX, ECX, EDX, Eflags
```

_vmNameShareInit calls Rmp_Alloc to initialize named shared RMP records segment and KSEMAAlloc to initialize named shared semaphores.

```
ENTRY    NONE
EXIT     NONE
USES     EAX, ECX, EDX, Flags
```

_vmDonateHighCode donates the unused portion of the last page of the HIGH2CODE and HIGH3CODE segments to the kernel resident heap.

```
ENTRY    NONE
EXIT     NONE
```

vmSGSAdd - Register given memory under Screen Group Switch control.
vmSGSRemove - Remove given memory from Screen Group Switch control list.
vmIsBusyArBMP - BMP busy function for arena records.
vmIsBusyObBMP - BMP busy function for arena records.
vmIsBusyCoBMP - BMP busy function for context records.
vmLinkAr - Link given arena record to object record's link list.
vmLinkCo - Link given context record to arena record's link list.
vmFindArCo - Walks linked arena and possibly context records belonging to an object until one with given owner and possibly address is found.
vmEachMapCheck - Check to see if worker needs to be applied to given Arena Record address.
vmEachMapSub - Walk AR chain, calling worker function if needed. This function converts a linear temporary buffer address to a selector:offset.

```
ENTRY    eax                - Temporary buffer linear address
EXIT     eax                - Selector:offset
          esi                - Points to selector
USES     eax, ecx, edx, esi
```

Perfview - mem queue length of active page faults mem count of page faults.
PvwCnt(mem,Recovered,0) - Perfview measurement hook.

```
TRACE MINOR = <PVWEI_Ct_mem_Recovered_0>,           //car01
TP          = @,                                     //car01
TYPE        = (PVW),                                 //car01
GROUP       = NOSTRICT                               //car01
```

PvwCnt(mem,Reassigned,1) - Perfview measurement hook

```
TRACE MINOR = <PVWEI_Ct_mem_Reassigned_1>,           //car01
TP          = @,                                     //car01
TYPE        = (PVW),                                 //car01
GROUP       = NOSTRICT                               //car01
```

Implementation

To be added.

Implementation Review

To be performed.

VM component Appendix

This page is intentionally left blank.

Glossary

linear address

On the 80386, a "virtual address" consists of a selector and an offset. The selector selects a descriptor in either the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT). This descriptor provides a base linear address to which the offset is added to produce the linear address to reference. This linear address is mapped through the page tables to a physical address. On most flat model paging processors, the selector level does not exist, and what Intel calls a linear address is called the virtual address.

0:32 address

On the 80386, a linear address; a flat model virtual address. The term is used to distinguish this kind of address from a 16:16 address, a 80286-style segmented address. The name is derived from the fact that the selector is implied and the offset is 32 bits long.

16:16 address

A 80286-compatible segmented virtual address. The term is used to distinguish this kind of address from a 0:32 address, a flat model virtual address. The name is derived from the fact that the selector is explicit and the offset is 16 bits long.

sparse object

An object to which no physical storage is committed at the time of allocation. VMAlloc(commit) must be called to obtain storage for regions of the object as desired.

type

Usually memory type, e.g. swappable, discardable, etc. The type of a page of virtual memory refers to the type of physical storage that the Page Manager has committed to supply for the page. The type of a region of a memory object refers to the type of storage that the Page Manager has committed to supply for all the pages in that region; the concept is meaningful only if all the pages are of the same type. Similarly, one may speak of the type of a memory object only if all the pages in that object are of the same type.

long-term lock

A lock whose duration is expected to be long, on the order of seconds. When a device driver requests a lock, it is always assumed to be long-term, since the system has no control over its duration.

short-term lock

A lock whose duration is expected to be short, on the order of milliseconds.

Size and Performance Considerations

To be added.

Implementation Estimates

	LOCs	Effort	ELOCs	MM
Add 32 bit address/offset support to all code, Linear addressed versions of MemMapxxxx (=> GetxxxxAddr)	200	1.0	200	0.8
Local Alias implementation	200	1.0	200	0.8
Change ptrace interface to accept linear address and size instead of selector. Rearrange local alias data structures into 2-D link list.				
Implement HUGE segments as a single segment	100	0.8	80	0.3
Expand SegLockInterface to 32 bits	80	0.8	64	0.25
New APIs	150	1.0	150	0.6
AllocMem, FreeMem, DosMemoryRegion				

ReallocMem				
SegSearch, SegAttach, SegDetach	40	1.5	60	0.25
Requires rewrite and support from Linear Memory Manager				
VMM Initialization	400	1.0	400	1.6
MoveDGROUPhigh				
VirtInit				
Initialize Memory Object Record table and Arena Record table. Build initial system arena, SysInit private arena, and initial shared arena. Provide interface to move objects in system arena.				
DevHlp support for new functions	150	1.0	150	0.6
Memory Object Record management	275	1.0	275	1.1
Mapping handles to object records, semaphore management, providing object information, address-to-handle mapping, context enumeration, arena walking.				
Object and Arena Table Management and Context List Management	125	1.0	125	0.5
Arena creation and deletion, record allocation and freeing, dynamic table growth.				
Freeing an Object	125	1.0	125	0.5
Allocating a New Object and Creating a New Reference to an Existing Object	205	1.0	205	0.8
Reallocating an Object	200	1.0	200	0.8
Locking an Area within an Object	30	1.0	30	0.1
Unlocking an Area within an Object	30	1.0	30	0.1
Changing the Type of Storage Committed to an Area within an Object	50	1.0	50	0.2
DevHlp Support for Existing Functions	45	1.0	45	0.2
Verify address of an object, allocate physical memory, free physical memory.				
Kernel Data Management	325	0.8	260	1.0
BMP, RMP, Kernel Resident Heap, Kernel Swappable Heap				
Determine Size of Largest Free Region	35	1.0	35	0.1
Discardable Object Support	45	1.0	45	0.2
Total	2810		2729	10.9

DOS Emulation

This page is intentionally left blank.

V8086 emulator Architecture

This page is intentionally left blank.

Problem Description/Objectives

This component includes the "pure" 8086 emulation support for the V86 mode of the 80386 (and 80486) CPU. It does *not* support DOS or hardware-specific emulation.

The V8086 emulator component provides the following:

- Software interrupt reflection support.
- IOPL sensitive instruction emulation support.
- I/O port trapping VDD service.
- I/O simulation support.
- Call when VDM interrupts enabled VDD service.
- Context hook services.
- Hook software interrupt VDD service.
- Change VDM execution flow services.

Software interrupt reflection is the mechanism used to service a INT instruction in V86 mode. Any time a software interrupt is executed in V86 mode, a GP fault is generated (if IOPL is less than 3). The GP fault handler code emulates the INT instruction "reflecting" the interrupt back into V86 mode. If IOPL is equal to 3, the software interrupt vectors through the corresponding Interrupt Descriptor Table (IDT) entry if the descriptor's DPL is equal to 3. If the IDT descriptor's is less than 3, the software interrupt is vectored to the GP fault handler. The CPU never uses the real mode interrupt vector table directly.

IOPL sensitive instruction emulation is needed when the CPU runs in V86 mode with IOPL < 3. Any instruction that can change the interrupt flag is considered "IOPL sensitive" and causes a GP fault in V86 mode. CLI, STI, LOCK, PUSHF, POPF, INT and IRET are these instructions. The I/O instructions (IN, OUT, etc.) are not IOPL sensitive in V86 mode; the IOPM (I/O privilege mask) in the TSS controls the I/O instructions. IOPL less than 3 can be used to "virtualize" the interrupt flag. The question is: when to do we virtualize the interrupt flag? If the interrupt flag is virtualized (IOPL < 3) all the time, performance suffers because the above instructions are emulated. If the interrupt flag is not virtualized (IOPL == 3), the stability of system is impacted because DOS applications can disable interrupts and loop thus halting the whole system.

In Cruiser, we run with IOPL == 3 to get maximum performance. To protect the system from a DOS application that disables interrupts for an extended period of time, we rely upon a hardware feature available in all known 80386 machines -- the "watchdog" timer. This is a programmable timer that will generate NMIs (non-maskable interrupts) at a periodic rate.

The watchdog timer will be used in conjunction with the real time clock interrupt to prevent a DOS application from disabling interrupts for too long a period of time (1 sec. at present). If a DOS application does disable interrupts for too long, the DOS application will be terminated, and the user notified via a popup.

At system initialization time, the watchdog timer will be enabled. Each time a real time clock interrupt occurs, the interrupt manager will clear the count of NMIs, and reset the watchdog timer. If the nmi handling code gets control, and the count is greater than the value indicating that one second has passed, the DOS application is terminated.

I/O port trapping is a service provided to VDDs in order to virtualize hardware devices. The IOPM in the TSS controls the GP faults on an I/O instruction. The GP fault handler calls the I/O port instruction decoder to route the trap to a VDD. Since all the faults for software interrupt reflection, IOPL sensitive instruction emulation and I/O port trapping are routed thru the GP fault handler in V86 mode, the GP fault handler's performance is critical.

Solutions/Justification

The OS/2 protection fault handler includes an instruction decoder that is executed when a V86 mode General Protection (GP) or Invalid Opcode fault occurs. This instruction decoder dispatchs to appropriate instruction emulation routines. The instruction emulation routines are broken into these categories:

- Software interrupts (INT 3, INT nn, INTO). These interrupts are "reflected" back into the VDM at the address pointed by the appropriate entry in the V86 mode interrupt vector table. The software interrupts with DPL 3 gates (INT 3, INTO) do not go the GP fault handler, but thru the normal interrupt gate and are reflected back into the VDM.
- IOPL sensitive instructions (CLI, STI, PUSHF, POPF, IRET). GP faults on these instructions only occurs when IOPL < 3. Faults on these instructions are used to "virtualize" the interrupt flag. The interrupt flag is virtualized (IOPL < 3) only when some VDD is waiting for VDM's interrupt flag to be enabled (via a VDH service). Normally IOPL == 3 and the interrupt flag is not virtualized. VPIC needs this service to detect when interrupts are enabled, to simulate an interrupt to the VDM.
- I/O instructions (IN, INS, OUT, OUTS). GP faults on these instructions are controlled by the TSS's IOPM. These faults are routed to the appropriate VDD by the I/O port trap service. If the type of I/O can not be handled by the VDD (like string dword OUTS), the

VDD can have the kernel simulate the I/O operation using a lower level I/O operation (word or byte OUT).

- Instruction overrides (LOCK, REP, REPNE, CS, DS, ES, SS, FS, GS, size, address). These overrides are tracked and passed to I/O trap handlers. The LOCK override is just ignored (the client's IP is incremented and the instruction after the LOCK is executed).
- Ring 0 privileged instructions (LIDT, LGDT, LMSW, CLTS, DRx, CRx, TRx). These instructions cause an invalid opcode (int 6) to be simulated into the VDM. HLT is a special case. If interrupts are enabled, the HLT causes the VDM to yield the CPU, and then continue running. This is compatible with the behavior of the 8086. On that chip, the HLT instruction halts the CPU until the next hardware interrupt occurs. If interrupts are disabled, then an int 6 is issued.
- Invalid V86 or real mode instructions (ARPL, LAR, LSL, VERR, VERW, STR, LTR, SLDT, LLDT). These instructions cause an invalid opcode (int 6) to be simulated into the VDM, except for certain flavors of the ARPL instruction, which is used by the "VDM breakpoint" service.

8086 Emulation is last in the invalid opcode (int 6) chain. If its handler gets control, a popup is displayed (noting that the VDM executed an invalid instruction) and the VDM is terminated.

The rest of the 8086 emulation is support that is not directly involved in instruction emulation and is basically services for VDDs and other components. These services can be broken up as follows:

- Exception traps (divide by 0, single-step, NMI, BOUND, invalid opcode). These V86 mode traps are not routed through the GP handler, but are received thru the corresponding IDT entry and are "reflected" back into the VDM.
- Context hook services. This is a mechanism to have a handler called during the "task time" context of particular VDM (local context hook) or any "task time" context (a global context hook). The VPIC VDD uses context hooks when it simulates a hardware interrupt to a particular VDM. This is a one-shot function; the handler is executed only once every time this function is called.
- Hook interrupt VDD service. VDDs are allowed to hook interrupts. The handler is called after the reflection, but before the ROM (post-reflection, pre-ROM). An IRET handler can be installed that is called after everything has executed (VDD handlers, VDM interrupt code, ROM) via another 8086 emulation service.
- Return to interrupt handler VDD service. This is a one-shot service used by the VPIC to execute the current VDM's interrupt code or by the instruction emulation to reflect back a software interrupt. This function edits the CS:IP on the kernel stack to point to the VDM's interrupt code. A return IRET frame is built on the client's stack from the original CS:IP (or with a VDM breakpoint if the IRET is to be trapped) and flags on the kernel stack.

V8086 emulator Design

This page is intentionally left blank.

Instruction emulation

The performance of the instruction decoder/emulator is critical to the overall performance of a VDM. In a running VDM, the frequency of faults that must be decoded can be 3000 per second. The first thing to do in the fault handler is to save all the client's registers and to set up a stack frame pointer. If the VM flag is set in the client flag register on the stack, the instruction decoder is executed. The first byte of the client's current instruction is fetched and decoded.

Once the instruction has been decoded, control is dispatched to an appropriate handler routine. Unsupported or invalid instructions cause an invalid opcode (INT 6) interrupt to be simulated into the virtual machine. The emulation routines are described in the following sections.

Software interrupts

The major functions of software interrupt emulation are:

- Software interrupt (INT x, INT 3, INTO) instruction trap handler.
- Hook interrupt (VDHInstallIntHook) service.
- Return to VDM interrupt handler (VDHPushInt) service.
- Hook return (VDHArmReturnHook) service.

Software interrupt instruction trap handler

The software interrupt number is retrieved from the byte after the opcode and the VDM's CS:IP is incremented past the INT opcode. "INT 3" and "INTO" have implicit interrupt vector numbers so no byte is fetched. Common interrupt "reflection" code (also used by the VDHPushInt function) is called with the interrupt number.

Hook interrupt (VDHInstallIntHook) service

The VDD interrupt handlers are kept as a linked list with the head in a 4K (400h * dword) table indexed by the interrupt number. Each time a VDD calls the hook interrupt service a block is allocated and put on linked list. This block contains the VDD handler address and the link to the next block.

The handler supplied to this service is a post-reflection handler. Post-reflection handlers are called after the INT instruction is reflected into the VDM interrupt code. A VDM breakpoint address has been put into the interrupt vector table entry at init time so control returns to the kernel at the end of the VDM's interrupt code chain. The VDD interrupt handler chain is then called. The ROM routine address (or whatever was initially in the interrupt vector table) that the VDM breakpoint replaced is returned to if the VDD handlers do not call VDHPopInt.

Return to VDM interrupt handler (VDHPushInt) service

VDHPushInt calls the interrupt reflection code to change the VDM's execution flow to the VDM's interrupt code. This is done by building a return IRET frame on the client's stack with the CS:IP and flags from the kernel stack frame. The VDM's CS:IP on the kernel stack is edited with the address from the VDM's interrupt vector table.

The VDHArmReturnHook service allows the IRET to be hooked after the VDHPushInt service is called. A VDM breakpoint is allocated and the address replaces the client's CS:IP on the return IRET frame. The client's CS:IP is saved away (see the figure).

When building the return IRET frame, the client's stack pointer may wrap around from 0 to 0ffffh. This is ok and does not terminate the VDM. If the client's stack pointer is equal to 1 when we start this function (before the VDD handlers are called), the VDM is terminated (to emulate what the 386 hardware does in real mode).

If the VDD installed an interrupt handler, a VDM breakpoint has been installed in the interrupt vector table before any VDM code hooks. After the VDM interrupt code chain has been executed, but before the ROM, the VDM breakpoint is executed. The VDD interrupt handlers are executed until one returns the "stop chaining" indication or the end of list is reached. The ROM code will be executed after the last VDD handler is called unless one of these handlers calls VDHPopInt. VDHPopInt emulates what the IRET does at the end of a v86 code interrupt handler popping off the IRET frame and restoring the VDM's CS, IP and flags to the IRET frame contents.

This VDHPushInt service is used by the software interrupt instruction trap and by VPIC in the last stages of simulating hardware interrupts.

Kernel Stk	Client stack	Kernel Stk	Client stack
+-----+		+-----+	+-----+

0	SS			SS	Flags
ESP				ESP	CS
EFlags				EFlags	IP
0	CS			Int CS	
IP				Int IP	
Before VDHPushInt			After VDHPushInt		

Kernel Stk	Client stack
0	Flags
ESP	VDMBP CS
EFlags	VDMBP IP
0	Int CS
Int IP	
After VDHArmReturnHook	
VDHPushInt/VDHArmReturnHook Services	

IOPL sensitive instructions

The major functions of IOPL sensitive instruction emulation are:

- STI, CLI, POPF, PUSHF or IRET instruction emulation.
- Interrupt flag virtualization.
- Call when interrupts enabled (VDHArmSTIHook) service.

IOPL-sensitive instruction emulation

STI and CLI emulation is simple; just set or clear the interrupt enable flag (either the virtual or real one). PUSHF has to retrieve the client's flags off the stack frame and to add them to the top of the client's stack. POPF has to retrieve the new flags from the first word on the client's stack and to put the new flags in the client's return stack frame. IRET is more complex, having to move the new CS:IP off the client's stack to the return stack frame. If there is a VDM breakpoint address on the IRET frame, control passes to the VDM breakpoint handler instead of wasting the time to return to V86 mode and back.

Both POPF and IRET clear the NT, RF and IOPL flag bits and set the VM flag bit. While the interrupt flag is being emulated the real IF flag is forced on. After the instruction has been emulated, if the virtual interrupt flag has been enabled, the VDD routines registered with the VDHArmSTIHook service for the current VDM are called.

When the IOPL sensitive instruction are being emulated, the "STI" instruction is going to be difficult to emulate properly. The "STI" is defined to enable interrupts AFTER the next instruction, but the 386 chip traps (when IOPL = 0) on the STI instruction itself. To properly emulate this instruction, the next instruction is single stepped or emulated before any pending interrupts are simulated to the VDM.

Windows/386 ignores this problem and simulates pending interrupts on the STI instruction itself. The chance that an interrupt is simulated at the wrong time is small because the IOPL sensitive instructions are emulated only if there is a pending interrupt to simulate and interrupts are disabled. Windows/386 has not found any applications that are broken by this behaviour. If it does cause a problem, the solution is to single step the next instruction after receiving the "STI" instruction trap and then to simulate any pending interrupts to the VDM.

Interrupt Flag Virtualization

Normally V86 code runs with IOPL equal to 3 and the IOPL sensitive instructions are not emulated and the interrupt flag is not virtualized.

The interrupt flag is left under the control of the DOS applications; otherwise, performance would suffer too much (as much as 300% in the testing Windows/386 has done). The only time IOPL-sensitive instructions are emulated (and IOPL is less than 3) is when the VDHArmSTIHook service is used by VPIC to wait until the interrupt flag is enabled.

Hardware interrupts can still be received while a VDM has its interrupt flag disabled. Even though the VDM has control of the real interrupt flag, a hardware interrupt can still be received during the I/O port trapping via software interrupt reflection handlers since they enable interrupts while processing these events. This means that a hardware interrupt can be received by a PDD and simulated to a VDM that has its interrupts disabled. This scenario demonstrates why the VDHArmSTIHook service is necessary.

Call when interrupts enabled (VDHArmSTIHook) service

When a VDD calls the VDHArmSTIHook service and interrupts in the current VDM are disabled, the VDM's IOPL is changed to 0 so the IOPL-sensitive instructions start being emulated. The VDD's handler is registered on a per VDM list. If the client's interrupts are enabled when this service is called, the VDD handler is called immediately.

This service is used by VPIC to delay a simulated interrupt until the VDM has enabled its interrupts.

I/O instructions

The major functions of I/O port trapping are:

- I/O instruction (OUT, IN, OUTS, INS) trap handler router.
- VDD port handler installation (VDHInstallIOHook) service.
- Enable/disable I/O port trapping (VDHSetIOHookState) service.
- Word, dword, and string I/O simulation.
- Provide access to ports that are not handled by VDDs

I/O instruction trap handler router

The I/O port instruction faults are handled by calling a VDD handler installed with VDHInstallIOHook. The port number is fetched from DX or from an immediate operand. If the port number is < 400H, a direct table lookup is used to find the VDD handler. The table is 4K (400H * dword) in size. If the port number is > 400H, a hashing scheme is used to save space. All the ports for a PC- or AT-like machine are below 400H. PS/2's allow ports higher than this, but the hashing scheme for these ports is adequate.

Most VDDs do not handle all types of I/O operations (input, output, byte, word, string, etc). These I/O operations are simulated by this component using lower level I/O operations. For example, string I/O has many variations (overrides, direction flag, input/output, size) and most VDDs don't support the various I/O types, so the I/O trap router breaks up the string I/O into byte or word I/O operations.

The install I/O handler service is passed a table of 5 I/O handler addresses instead of one I/O trap handler address. The table has a handler for the 4 most common I/O types and one handler for everything else:

```
byte input
byte output
word input
word output
```

other (dword input/output, string I/O, etc)

VDD port handler installation (VDHInstallIOHook) service

This service installs a VDD handler for a specified I/O port. If a port has already been hooked by another VDD, an error is returned. Only one VDD may install a handler for a particular port. If the port is < 400H, the port/handler table is updated. Otherwise, a hash table node is allocated and linked into the appropriate hash table list head. This service automatically enables a port trap for the specified port globally for all VDMs.

This service is used by every VDD that controls I/O port access.

Enable/disable I/O port trapping (VDHSetIOHookState) service

This service enables and disables I/O port trapping. This involves clearing or setting the appropriate bit(s) in the IOPM. One of the major problems is the size of the IOPM. For an IOPM that contains the ports in the range 0 - 0ffffh, it would be 8K (1 bit per port).

A separate IOPM needs to be maintained for each VDM. This is implemented by having every VDM process use its own TSS and having the VDM's IOPM at the end of this TSS. Protected mode processes will still use the one common TSS. The initial IOPM starts out at 400H ports (128 bytes), but is grown (since it is at the end of the TSS object) if a VDD disables trapping on a port > 400H. If a port greater than the size of IOPM has trapping enabled (via VDHSetIOHookState), the IOPM is not grown because the default behavior of the CPU is to trap the port. If a port needs to be enabled or disabled globally, VDHSetIOHookState must be called for each VDM.

This service is especially useful to the video VDD. When a DOS application is running full-screen in the foreground, it will run faster if no I/O port trapping is done.

To avoid trying to grow the IOPM at the time of a VDHSetIOHookState call (which could fail if there is not enough memory), VDHSetIOHookState may be called only for ports which have been hooked *without* using the VDHII_ALWAYS_TRAP flag on the VDHInstallIOHook call.

VDHInstallIOHook will grow the IOPM if necessary when the VDHII_ALWAYS_TRAP flag is not present. Since VDHInstallIOHook is generally called at VDM creation, this makes it easy for a VDD to handle the case where the call fails due to lack of memory (the VDD would most likely fail VDM creation). If the VDD got an error from VDHSetIOHookState, there would be no sensible action for it to take.

Access to ports not trapped by a VDD

In order to support DOS applications that use hardware that does not have VDDs, it is necessary to provide a mechanism for DOS applications to access ports on those devices. This is done by having the default handler for each IO port turn off trapping for the port in question. All accesses after the first will go straight through to the hardware. This will only be done for ports that are not trapped by a VDD.

Global and local context hooks

Global context hooks are a mechanism to delay work until the CPU is at task time (as opposed to interrupt time). If a global context hook is

"set" at task time, the handler is called immediately. Global context hooks are similar to the Ctrl-C or Ctrl-Break interrupt time signal posting. When a Ctrl-C is received at interrupt time (from the keyboard device driver), a global flag is set indicating pending signals. This flag is checked on the way out of the kernel (ExitKmode) and by a separate context hook thread that is woken up any time there are pending global context hooks and the system is in the idle loop.

Global context hooks are implemented by checking a global flag every time we leave the kernel. If the flag is set, a global variable (which could be the flag) contains the head of a linked list of global context hook handlers. The global context hook service is used by VPIC VDD when simulating an interrupt, and by several of the VDDs that need to get to task time from interrupt time.

Local context hooks are a mechanism to delay work until the CPU is at task-time in the context of a specific process. When a VDD sets a local context hook, that hook is guaranteed to execute before any user code (i.e., code executing in V86 mode).

Local context hooks are implemented by checking a per-thread flag when the thread leaves the kernel. If the flag is set, a linked list of local context hook handlers is called. This is a more general "force flag" mechanism used in the OS/2 kernel today. Local context hooks can be set at interrupt time.

Imported Interfaces

8086 Emulation uses various kernel and VDH services. See the sources for details.

Design Constraints

- Only one VDD may install a handler for a particular I/O port.

V8086 emulator Implementation

Instruction Decoding

There are several ways to decode the instruction:

1. Directly index a table of emulation routine addresses with the opcode.

+

This is about the fastest way to decode the instruction. The following code fragment takes 16 cycles:

```
movzx    ebx,byte ptr [esi]      ; 6  (EBX) = inst opcode
jmp      OpTable[ebx*4]          ; 10 jump to inst emulation rout
```

-

Costs 1K (256 dwords) in memory.

2. Use the high nibble to index an array of second level table addresses (this table is 16 dwords). The low nibble is used to index the table found above (which is also 16 dwords). This saves space because the second level table entry may not have to point to anything if the instructions with that high nibble are invalid.

+

Saves space. 64 bytes (first table) + 64 * (9 second tables) = 640 bytes

-

Slower because the high nibble is shifted and a second table indexed. The following code fragment takes 27 cycles:

```
movzx    ebx,byte ptr [esi]      ; 6  (EBX) = inst opcode
```

```

mov     eax,ebx                ; 2 (EAX) = inst opcode
and     eax,0fh                ; 2 (EAX) = low nibble
shr     ebx,4                  ; 3 (EBX) = high nibble
mov     ebx,OpTable[ebx*4]     ; 4 (EBX) = second table addr
jmp     [ebx+eax*4]            ; 10 jump to inst emulation rout

```

3. First index an array of bytes with the opcode. The contents of a byte in this array is the offset into a second table of emulation routine addresses.

+

Saves space. 256 bytes + 4 * (41 emulation routine addresses) = 420 bytes

-

Slower because the second table needs to be indexed to get to the routine address. The following code fragment takes 20 cycles:

```

movzx   ebx,byte ptr [esi]     ; 6 (EBX) = inst opcode
mov     bl,OpTable[ebx]        ; 4 (EBX) = routine addr index
jmp     RoutTable[ebx*4]       ; 10 jump to inst emulation rout

```

Interrupt Hooking

There are three possibilities for hooking interrupts:

1. Call the VDD software interrupt handlers first when the INT instruction fault is received (pre-reflection). The VDDs have the choice whether to eat the software interrupt or allow it to be reflected into the VDM. After the VDDs had their crack at the interrupt, it is reflected to the VDM's handlers. This is what Windows/386 does.

+

Always allows the VDD to have first crack at the interrupt.

+

VDD gets control faster if it is called before the VDM. The video VDD speeds up INT 10 write tty requests and doesn't want to pay the cost going thru the interrupt reflection first.

-

A "pushf" flags and "call" of the address in the vector table are not be trapped. Only the INT instructions are trapped. Windows/386 has not found any app that does this on the first invocation. They assume that the "pushf" flags and "call" is only done to chain to the next VDM handler.

-

This breaks compatibility. If a TSR intercepts INT 10 to see all the write tty commands to detect a scroll up and copy the top line for scroll buffering and a VDD eats all the INT 10 write tty commands, this TSR is broken.

2. When an INT trap is received, it is first reflected into the VDM interrupt code (post-reflection). A VDM breakpoint address has been put into the interrupt vector table entry at init time so control returns to the kernel at the end of the VDM's interrupt code chain. The VDD interrupt handler chain is then called. The ROM routine address that the VDM breakpoint replaced is returned to if the VDD handlers didn't "eat" the interrupt.

+

If the address in the interrupt vector table is called with a "pushf" flags and "call" the VDM interrupt handler chain are called exactly as if an INT instruction was used.

+

Compatibility is maintained because the VDM get the first crack at software interrupts.

-

VDD may need to receive control first on a software interrupt. I am not sure of all the cases where VDD need to be first yet, but this is the major reason Windows/386 went with #1.

3. Allow the VDD the option of "pre-reflection" handlers (#1) or "post-reflection" (#2) when it hooks a software interrupt. Pre-reflection handlers should not (but can) eat the software interrupt to maintain compatibility. Pre-reflection handlers allow the video VDD to stop video updates while in the INT 10. Post-reflection handlers would be where the video VDD intercepts write tty INT 10's to speed them up.

+

If the address in the interrupt vector table is called with a "pushf" flags and "call" the post-reflection handlers are called.

+

Compatibility is maintained because VDMs get the first crack at software interrupts.

Cruiser implements the second method, i.e., we ALWAYS reflect the interrupt to the real-mode interrupt vector table. A VDD's interrupt hook gets control by virtue of the trap on the VDM breakpoint. As an optimization, the interrupt reflection code tests to see if the real-mode interrupt table vector contains a VDM breakpoint. If it does, then we stay in protected mode and dispatch to the correct handler.

Interrupt Flag Virtualization

There are two possible choices for interrupt flag virtualization:

1. Run with IOPL = 0 and always emulate the IOPL sensitive instructions. This allows virtualization of the VDM's interrupt flag and the

real interrupt flag can always be enabled. The instruction emulation routines track the VDM's interrupt flag state separately from the real interrupt flag.

- + VDM's cannot crash or affect the rest of the system by disabling interrupts.
- There is a performance penalty when emulating IOPL sensitive instructions. There are few programs like BASIC that execute a lot of IOPL sensitive instructions affecting its performance. Most programs don't execute a lot of these instructions and the I/O port and software interrupt trapping seem overshadow the IOPL emulation time.

2. Run with IOPL = 3 and no interrupt flag virtualization. This leaves the control of the real interrupt flag in the hands of DOS applications. The instruction emulation routines update the real interrupt flag in the client's stack frame.

- + VDM's don't have to pay the price of IOPL sensitive instruction emulation. This may be a significant performance penalty depending on how often the application executes the IOPL sensitive instructions.
- A DOS application can crash the whole system by disabling interrupts and spinning in a loop. If the hardware has a watchdog timer, this crash could be detected and the VDM terminated.
- Even if there is a watchdog timer to prevent complete interrupt lock-out, DOS applications can still adversely affect the rest of the system by disabling interrupts too long.

I/O Port Trapping

There are a few possibilities for the method that the I/O operation type (input/output, byte/word/dword, string, rep, addr32, reverse) is passed a to the VDD and the mechanism for the VDD to indicate that this I/O needs to be simulated:

1. The I/O port handler is passed byte containing the I/O type. A simple compare and jump can filter off the I/O operations the VDD can handle (usually byte input/output). The VDD can return some indication (like carry set) that the I/O needs to be simulated. This is the method Windows/386 uses (though they actually jump to a simulate I/O routine from the VDD instead of returning some indication).
 - + In real world VDDs common entry points for all the I/O types are used in most cases. There is usually some pre and post processing common to all I/O types and even for different port numbers. The COM VDD is a good example, it has two different sets of I/O port routines; one for the COM port owner and one for non-owners.
 - + Uses less memory.
 - Getting to the port handler is slower because of the compare and jump to filter off the I/O types not accepted by the VDD. The VDD also decodes the type flag if it does handler more than the usual byte input/output.
 - I/O simulation is slower because the I/O type is decoded to execute the appropriate simulation routine.
2. The install I/O handler service is passed a table of 12 I/O handler addresses instead of the one I/O trap handler address. The table has a handler for each general I/O type:

byte input	byte output	byte string input	byte string output
word input	word output	word string input	word string output
dword input	dword output	dword string input	dword string output

The string I/O operations are still passed a flag containing the rep, addr32, segment override and direction flag state. Having a separate entry in the table for each of these variations would be too costly. For the I/O types not accepted by a VDD, no handler needs to be installed in this table (zero instead). The install service can then put the address of the appropriate I/O simulation routine in the zero'ed entries.

- + Dispatching to the I/O handler is faster in some cases. Most VDDs support only byte I/O type.
- + I/O simulation is faster because the simulation routine is dispatched to directly.
- Uses memory (48 bytes per port). There are approx. 100 ports to be trapped overall: 4800 bytes.

3. The install I/O handler service is passed a table of 5 I/O handler addresses instead of one I/O trap handler address. The table has a handler for the 4 most common I/O types and one handler for everything else:

byte input
byte output
word input
word output
other (dword input/output, string I/O, etc)

- + Dispatching to the I/O handler is faster in most cases. Most VDDs probably only support byte I/O type.
- + I/O simulation is faster because the simulation routine is dispatched directly to.
- Uses memory (20 bytes per port). There are approx. 100 ports to be trapped overall: 2000 bytes.

Enable/Disable I/O Port Trapping

A few possibilities for enable/disable I/O port trapping:

1. Have one enable and one disables function that works for all ports. This can be implemented by putting a TSS in every VDM's PTDA and having the VDM's IOPM at the end of its PTDA. The initial IOPM can start out at 400H ports (128 bytes), but is grown (since it is at the end of the PTDA) if a VDD disables trapping on a port > 400H. This depends on VDM PTDA's being movable so the PTDA has to be moved out of the VDM address space (and all the DOS emulation code converted to run in protected mode). If a port greater than the size of IOPM is enabled (via VDHSetIOHookState), nothing happens because the default is to trap the port. If a port needs to be enabled or disabled globally, the "VDHEnumerateVDMs" function can be used.
 - + Faster task switching, no IOPM coping required (approx. 80% faster).
 - The IOPM could grow to 8K if a VDD disables trapping with a high port address.
 - Costs 104 bytes and a GDT selector per VDM PTDA.
2. Have local and global enables and disables. Local enables/disables work on a per VDM basis with ports < 400H. Global enables/disables work on any port address, but every VDM enables or disables the port. This keeps the per VDM IOPM small (128 bytes) so a lot of memory isn't copied on task switches.
 - A VDM IOPM (128 bytes) copied during task switches (takes 135 cycles or 270 cycles if switching from a VDM to another VDM).
 - Local enables/disables of ports > 400H are not easy. Every time a VDM is scheduled the port can be enabled or disabled globally.

DOS Properties

This page is intentionally left blank.

Advanced Properties Requirement

This page is intentionally left blank.

Problem Description/Objectives

Competitor products (Windows/386, DesqView/386, VM/386, VP/ix) provide the ability to control special properties for DOS applications. These include (but are not limited to):

- DOS Memory Size (0..640K)
- LIM Memory Size
- XMS Memory Size
- Idle Detection Threshold

Justification

DOS sessions have many more configurable properties than do PM and VIO sessions: DOS memory size (up to 640K), LIM memory size, XMS memory size, DOS device drivers, etc.

In order to be competitive with other operating environments that support multiple DOS sessions (Windows386, VM/386, DesqView, VP/ix), we must allow the user to configure these properties.

Functional Characteristics

Provide VDMPropertyDialog service in the VDM Shield layer (PMVDMP.DLL)

Alternative Solutions

The shell team proposed that the VDMPropertyDialog code be responsible for manipulating the EA directly. This is undesirable, since the PM Shell needs to read this EA in order to pass the information on to SheStartLongProgram.

The solution described in the SDD pages below allows the PM Shell to treat the DOS properties as a single chunk of bytes, and nicely separates the storage mechanism from the user interface.

System design documentation

The following description of the PM Shell behavior is brief, and included only to give context for the discussion of the Advanced Properties dialog. Please see the PM Shell specification for exact details.

The PM Shell stores session startup information in Extended Attributes (EAs) attached to program files, data files, and "reference" files. Let us refer to these as "session profiles". A "reference" file is an empty file whose only purpose is to record session information. This allows a user to set up several different sessions which use the same program.

The PM Shell has one or more dialogs that are used to create and edit these session profiles. In the lowest level dialog for editing a session profile, there will be an "Advanced..." button. This will appear only when editing a DOS session profile.

As far as the PM Shell is concerned, the DOS Properties are a chunk of bytes:

- They are stored in a separate EA, whose name is at the discretion of the PM Shell (one possible name is DOSPROPERTIES).
- When a DOS session profile is created, the PM shell creates an empty (i.e., zero length) set of DOS Properties.

- When a user wishes to modify the DOS properties, the PM Shell calls VDMPropertyDialog, passing in the (possibly empty) set of existing DOS properties.
- When VDMPropertyDialog returns successfully, the PM Shell will receive a modified set of properties, and it must store them into the EA, over-writing the previous properties.
- When the Shell starts a DOS session, it reads the EA into memory, and passes a pointer to these bytes in the PPROGDETAILS.pszEnvironment field to SheStartLongProgram.

```

/**EP+ VDMPropertyDialog - PM Shell DOS Properties Dialog
*
* This function displays a Dialog box that allows the user to
* modify the VDM properties.
*
* ENTRY   hwndClient - hwnd of top-level window of shell
*          hwnd       - hwnd of parent dialog box
*          npchIn      - buffer containing existing VDM properties
*                      If NULL, default values are chosen for all
*                      VDM properties.
*                      If not NULL, then default values are chosen
*                      for any VDM properties not in the list.
*          cbIn        - size of npchIn buffer
*          ppchOut      - pointer to receive buffer address
*          pcbOut       - pointer to receive size of buffer
*
* EXIT-SUCCESS
* returns TRUE
* ppchOut filled in with address of a buffer containing the
* new VDM properties.
* NOTE: The caller must free this buffer with DosFreeSeg.
* pcbOut filled in with size of the ppchOut buffer.
*
* EXIT-FAILURE
* returns 0
* User cancelled operation, or some other failure
* occurred. ppchOut and pcbOut contents are undefined.
*/
USHORT EXPENTRY VDMPropertyDialog(HWND      hwndClient,
                                  HWND      hwnd,
                                  PSZ       pchIn,
                                  USHORT    cbIn,
                                  PSZ FAR *ppchOut,
                                  PUSHORT   pcbOut)

```

Advanced Properties Overview

Virtual Device Drivers (VDDs) call VDHRegisterProperty to register their properties, and call VDHQueryProperty at VDM creation to get the value of a property.

When the user is setting the properties for a DOS application, the PM Shell calls DosQueryDOSProperty to enumerate all the registered properties and their attributes. This information is used to display a simple dialog box that allows the user to view and modify the DOS properties.

For a running DOS session, the Shield layer adds a "Properties..." menu to the system menu. Selecting this menu causes a dialog box to appear, whose appearance is very similar to that of the PM Shell. Unlike the PM Shell dialog, however, this dialog allows the user to change a property (with DosSetDOSProperty) while the DOS session is running (only the set of properties that can be changed are presented).

- Properties can be BOOL, INT, STRING, or ENUM.
- Properties have "bounding" information (min,max,step for INT; string length for STRING, list of valid string values for ENUM).
- Properties have validation functions (only for STRINGs) and set functions (for any type).
- VDDs "register" a property with VDHRegisterProperty
- VDDs query a property value with VDHQueryProperty

- The PM Shell uses DosQueryDOSProperty to enumerate all registered properties, and to get bounding information. These properties are stored along with the common session startup information in OS2.INI (or in a reference file EA, for the collapsed shell).
- The VDM Shield layer (Settings... submenu in the system menu for a DOS session) uses DosQuery/SetDOSProperty to change a subset of the properties for a running VDM. This can be used to tune the idle detection setting, for example.

See the section below for details on these APIs and their usage.

Performance Specification

With advanced properties, we will be able to provide tuning control over heuristic code that detects idle DOS applications. This will allow the user to reduce the wastage of CPU time by idle DOS applications.

Usability Characteristics

The advanced properties give a user the ability to exercise more control over the consumption of system resources by a DOS application. At the same time, this control is not immediately obvious to the unsophisticated user.

NLS Considerations

All strings are installable components, and hence are easily changed. Furthermore, help support is designed in, so that only help files need be modified.

DOS Compatibility

This improves our DOS compatibility, since providing adjustable properties allows a user to configure a DOS session more flexibly, as may be necessary to support certain applications that do not work (or do not work well) in the default configuration.

Dialog Box Layouts

The current property being manipulated is selected using a "drop-down list box". The value is presented using a control that corresponds to the property type:

BOOLEAN	Check box	INTEGER	Coupled single-line edit control and scroll bar	STRING	Single-line edit control	MLSTRING
	Multi-line edit control	ENUMERATION	Drop-down list box			

In the diagrams, the various controls are represented as follows:

Property:	DOS memory size			V
Value:	512	<	#	>
<div> <div>Ok</div> <div>Default</div> <div>Cancel</div> <div>Help</div> </div>				

STRING Property

Advanced DOS Properties				
Property:	Shell			V
Value:	c:\os2\mdos\command.com			
<div> <div>Ok</div> <div>Default</div> <div>Cancel</div> <div>Help</div> </div>				

MULTI-LINE STRING Property

Advanced DOS Properties				
Property:	DOS device drivers			V
Value:	c:\mydos\dev\ansi.sys			
	c:\dev\conedit.sys			
			#	
			V	

< #		>	
/-----\	/-----\	/-----\	/-----\
Ok	Default	Cancel	Help
\-----/	\-----/	\-----/	\-----/

BOOLEAN Property

Advanced DOS Properties	
Property:	Video modes V
Value:	Text only V
/-----\	/-----\
Ok	Default
\-----/	\-----/

DOS Session Customization via Advanced Properties

DOS sessions have many more customizable properties than do OS/2 sessions. MVDM provides a common mechanism that supports both a standard set of properties, and allows Virtual Device Drivers (VDDs) to register custom properties.

The Standard Properties are a subset of the configuration settings available in the CONFIG.SYS file of DOS, plus some additional settings that are required for MVDM.

- BREAK - BOOLEAN - controls Ctrl+C checking in the INT 21 path
- FCBS - INTEGER - controls reuse of internal file system data for DOS applications that use FCBs in an ill-behaved manner.
- DEVICE - STRING - specifies DOS character device driver(s)
- SHELL - STRING - specifies command interpreter.
- RMSIZE - INTEGER - specifies size of DOS memory arena.

The primary reason for these properties is that DOS applications are not as careful about consuming resources.

Overview of Properties Data Flow

The following scenarios describe how properties are created, modified, inspected, and destroyed. Each of them is described in detail below.

- Property Registration
- Properties at VDM Creation
- Changing properties in the PM Shell
- Changing properties for a running VDM.

Property Registration

At VDD initialization, VDDs register any properties they need. This information is stored in a "database" in the kernel, and used to support all property-related operations.

The following information is registered for each property:

- Name

The property name presented to the user. This may contain blanks, and related properties should have common prefixes so that they sort together in the user interface (i.e., "Printer buffer size", "Printer timeout", "Printer automatic close").

- Ordinal

For the "standard" properties, specific ordinals are used so that the kernel obtain the value independant of the name string. See the VPORD definition.

- Help File

The name of the help file containing help information on this property.

- Help ID

The help ID of the main topic for this property.

- Type

The property type. Boolean, integer, enumeration (list of valid strings), single-line strings, and multi-line strings are supported. This allows the user interface to display an appropriate control for each property. See the VPTYPE definition.

- Flags

These control aspects of the property. In particular, whether the property can be changed while a VDM is running.

- Default Value

If the user does not supply a value, this default value is used.

- Validation information

This information allows the user interface (and the kernel) to validate property values without bothering the VDD. For strings, this is the maximum string length. For integers, this is the minimum, maximum, and step values. For enumerations, this is the list of valid strings.

- Function

This function is used for validating string properties, and for notifying the VDD when the user has changed a property value for a running VDM.

Please see VDHRegisterProperty for details on these fields.

Properties at VDM Creation

On the DosStartSession call, the Environment pointer points to a properties buffer when creating a DOS session. This buffer contains the buffer length, followed by one zero or more properties. For each property, the property type, name, and value are specified.

The kernel parses the property buffer to create initial values for these properties. Default values are taken for any registered properties not specified on the DosStartSession call.

For any properties which have not been registered, the information in the buffer is ignored. This allows the system to run without errors in the case where a VDD that registered a property is not loaded (config.sys was changed), and yet the PM Shell had saved a value for the property.

After these per-VDM properties are initialized, the VDD VDM_CREATE hooks are called. At this point, the VDDs may call VDHQueryProperty to get the values of their properties.

See the PROPERTYBUFFER definition for the format of the buffer passed on the DosStartSession call.

Changing properties in the PM Shell

The PM Shell can be used to set up information used to run OS/2 and DOS applications. For DOS applications, there is an "Advanced..." button in one of the "Properties..." dialog boxes. Pressing this button calls up the "Advanced DOS Properties" dialog box. The user can manipulate the property values (which are set to the default values the first time this dialog is invoked for a specific entry in the PM Shell), and then save them.

The Advanced Properties dialog box uses the DosQueryDOSProperty API to get the list of property names and detailed information on each property. It uses the DosSetDOSProperty to validate string properties.

See DosSet/QueryDOSProperty for more information on these services.

Changing properties for a running VDM.

The Shield layer inserts a "Properties..." menu item on the system menu for all DOS sessions. Selecting this menu item causes a Properties dialog box to be displayed, which allows the user to modify properties for a running VDM. Only those properties that have been registered as "modifiable at run-time" (see VDMP_CREATE) are displayed for modification.

The DosSet/QueryDOSProperty services are used to query the current value and set new values.

DOS emulation Architecture

This page is intentionally left blank.

Problem Description/Objectives

The objective of this component is to provide DOS services in a manner that provides maximum compatibility for applications which run under IBM DOS 4.00.

Other components provide the hardware and VMCB environment needed for compatibility. The DOS emulation component extends this support to provide the same additional environment characteristics provided by IBM DOS 4.00.

Structural Overview

DOS emulation (DOSEM) is achieved by having a very small DOSEM kernel which runs in V86 mode, and a much large portion of code that runs in protected mode. The V86 code serves several purposes:

1. It supports console input polling, without forcing the OS/2-386 kernel to take on the reentrancy problems of the DOS kernel.
 2. It allows us to leverage existing DOS source to ensure compatibility and reduce the amount of rewriting required.
 3. It gives us the flexibility to move functionality in and out of V86 mode to tune for size and speed.
-

Supported DOS Features

The DOS emulation component supports the documented aspects of the following DOS features:

- INT 20h Program Terminate Interface
- INT 21h System Call Interface
- INT 22h/INT 23h Terminate and Ctrl-Break Address Interfaces
- INT 24h Critical Error Handler Interface
- INT 25h/INT 26h Absolute Disk Read/Write Interfaces
- INT 27h Terminate But Stay Resident Interface
- INT 28h Idle Loop Interface
- INT 2Fh Print Spool Interface
- DOS Console Device Driver
- DOS Device Driver Loading/Support
- DOS Program Loading and Execution
- DOS FCB I/O Support
- DOS Memory Manager
- DOS NLS Support
- DOS PDB Process Support
- VDM Entering and Exiting Kernel Mode Support
- Special DOS Compatibility Mechanisms

In addition, some of the undocumented aspects of these features (especially INT 21h) are supported because a large number of significant DOS applications rely upon these interfaces.

DOS Compatibility

"DOS compatibility", as it is known in the marketplace, is an attribute applied to a hardware/software combination that runs, or attempts to run, DOS applications. Since most of these combinations include a standard release of DOS, the only variability is in the hardware and ROM BIOS of the machine.

The DOS emulation component only addresses the DOS aspect of this compatibility equation. The VDMM and VDDs work together to provide hardware and ROM BIOS compatibility.

OS/2-386 is at least as compatible as 286 DOS-boxes when it comes to running DOS applications. In particular, because PDDs (PDDs) are loaded above 1Mb and only the DOS emulation code needs reside below 1Mb, there will be in excess of 600K free memory for DOS applications. This exceeds even DOS itself.

There are some DOS applications and products that cannot be supported, due to the nature of the DOS emulation and the multi-tasking and protection demands of OS/2-386:

- Windows 1.0x (has internal DOS structure dependencies)
 - MS/PC NET (has internal DOS structure dependencies)
 - Norton Disk Utilities (doesn't work with multi-tasking disk I/O)
 - DOS block device drivers (don't understand multi-tasking)
-

DOS emulation Initialization

This page is intentionally left blank.

Initialization and VDM Creation

DOS emulation component initialization is divided into two stages. The first occurs during OS/2 system initialization. The second stage occurs for each VDM as each is created.

OS/2 Initialization Stage

The DOS emulation component is involved in OS/2 system initialization, because it needs access to information that is contained in the CONFIG.SYS file. As the OS/2 initialization procedure processes the CONFIG.SYS file, it records parameters which are related to the 3XBOX for the DOS emulation component. This information includes information on the "FCBS" command, "RMSIZE", and the "DEVICE" commands that specify 3XBOX device drivers. The CONFIG.SYS values become the defaults for the standard DOS properties for all VDMs.

Here it is important to note that LASTDRIVE parameter of config.sys, which was a no-op in earlier versions of OS/2, is used by this component. In older versions of OS/2, lastdrive was always hardcoded as z. That means DOS apps like NOVELL network etc. wouldnt get any dos drive. But as we are supporting these types of apps now, we have to reserve few drives for exclusive use by a VDM. This is what lastdrive command will do, to get few drives for VDMs exclusive use. The default value for this command will be z incase it is not specified in config.sys. This makes it same as todays operations where all the drives are reserved for OS/2. Only those users who have to run apps like NOVELL should use this command.

No other processing related to the DOS emulation component occurs at OS/2 initialization time. In particular, it is important to note that 3XBOX device drivers are not loaded nor initialized in this stage. This stage occurs even before the initialization of the VDMvirtual device drivers.

VDM Creation Initialization Stage

After the VDM manager calls the VDD creation-time initialization routines, it passes control to the DOS emulation. At this point, the V86 memory organization appears as in The figure.

```
-----
>
>          VMCB ROM
>
-----
>
>          Video H/W
>
-----
>          EBIOS Data
-----
>
>
>
>
>          Unassigned Memory
>
>
>
>
```

```

-----
>      VDD Assigned Memory
-----
>
>      VDOS Kernel Code
>      and Data
>
-----
>      DOS Communication Area
-----
>      VMCB Data
-----
>      INT Vector Table
V86 Memory Before DOS emulation Component Initialization

```

The DOS emulation component performs these steps:

1. Initialize VDM-Related Kernel Structures

Certain structures in the OS/2 kernel are initialized in preparation for processing VDM requests. Namely, the System File structures which are used for FCB I/O are allocated and initialized. Other miscellaneous structures are also initialized.

2. Load DOS Emulation Kernel (DOSKRNL)

Code which runs in V86 virtual memory is loaded at the high end of the VDM memory address space. This is a DOS *.COM file.

3. Start Virtual Processor Mode

The protect-mode initialization code returns to the VDM manager which passes control to initialization code in the virtual area. This represents the first transition to virtual mode, when memory is organized as in The figure.

```

-----
>
>      VMCB ROM
>
-----
>
>      Video H/W
>
-----
>      EBIOS Data
-----
>      VDOS Init Code
-----
>
>
>      DOS Memory Arena
>
>
>
>
-----
>      VDD Assigned Memory
-----
>
>      VDOS Kernel Code
>      and Data
>
-----
>      DOS Communication Area
-----
>      VMCB Data
-----
>      INT Vector Table
V86 Memory at Start of V86 Processing

```

4. Open Standard Devices

The initial 5 file handles (stdin, stdout, stderr, stdaux, and stdprn) are opened.

5. Initialize Virtual Device Driver DOS Device Driver "stubs"

Some VDDs provide a DOS device driver "stub". By the time DOS emulation gets called, these "stubs" have already been inserted into the V86 memory. This step involves calling the inserted initialization code and linking the devices into the device chain. Unlike real DOS device drivers, however, the return from the initialization does not allow reducing the size of the driver code.

6. Initialize DOS Device Drivers

Each device driver specified in the OS/2 CONFIG.SYS file is loaded into the VDM and initialized. Following that, any DOS device drivers

passed on the DosCreateVDM call are loaded and initialized. This is performed one at a time to allow the device drivers to consume only the memory that they require or to de-install themselves entirely.

As each device is initialized, it is added to the chain of devices in the 3XBOX

During initialization, the device drivers may issue a limited set of INT 21h system calls: functions 01h through 0Ch, 25h, 30h, and 35h. This restores functionality that had been removed from prior versions of OS/2.

NOTE: The result is UNDEFINED when a DOS device driver issues an INT 21h system call other than those described above. This is consistent and compatible with DOS. At worst, issuing an unsupported INT 21h system call will crash the VDM. [Ed. Note: In the current sources, most, but not all, INT 21h system calls seem to work. To avoid unnecessary testing, however, we do not claim to support anything beyond what DOS supports.]

After all device drivers have been initialized, the initialization code is discarded.

7. Load and Execute 3XBOX Shell

The shell that was specified in the SHELL command in the CONFIG.SYS file is loaded, the initialization code in the V86 memory arena is discarded, and control is passed to the shell program. The SHELL specified in CONFIG.SYS can be overridden on the DosCreateVDM call. This may be a useful feature if an OEM wishes to allow different nationalities of COMMAND.COM, for example. V86 memory is then organized as shown in The figure.

```
-----
>
>          VMCB ROM
>
>
>-----
>          Video H/W
>
>-----
>          EBIOS Data
>-----
>
>
>
>          DOS Memory Arena
>
>
>
>-----
>
>          COMMAND.COM
>
>-----
>          DOS Device Drivers
>-----
>          VDD Assigned Memory
>-----
>
>          VDOS Kernel Code
>          and Data
>
>-----
>          DOS Communication Area
>-----
>          VMCB Data
>-----
>          INT Vector Table
V86 Memory at Start of Shell Execution
```

Before passing control, the PDB of the shell is initialized with the command line parameters as specified in the CONFIG.SYS file. As an extension to IBM DOS 4.00, after these parameters, an additional string is appended, separated from the command line string by a NUL byte, which specifies the drive and directory of the VDOS after the shell completes its initialization.

This extension allows a starting drive and directory for real mode applications as is provided for protect mode applications using the Page Manager.

Kernel Mode Transition

The OS/2-386 3XBOX isolates some of the code that is dedicated to the 3XBOX from the OS/2 kernel by putting it into the 3XBOX

memory area. Instead of code which runs in both DOS and OS/2 modes, switching between them freely, the OS/2-386 3XBOX restricts procedures to one mode or the other. Furthermore, only a limited number of entry points in the OS/2 kernel may be called from the 3XBOX.

When the DOS kernel needs OS/2 kernel services, it specifies the kernel procedure, by an index, and executes a CPU instruction that causes the V86 manager to trap and transfer control to the specified kernel procedure. The section on the V86 manager describes this method in further detail.

Using this method, the code which is located in the V86 address space is strictly for support of the VDM. In particular, all memory arena services are in this area. To provide as much application space in the VDM as possible, however, the size of this VDM-only code is minimized.

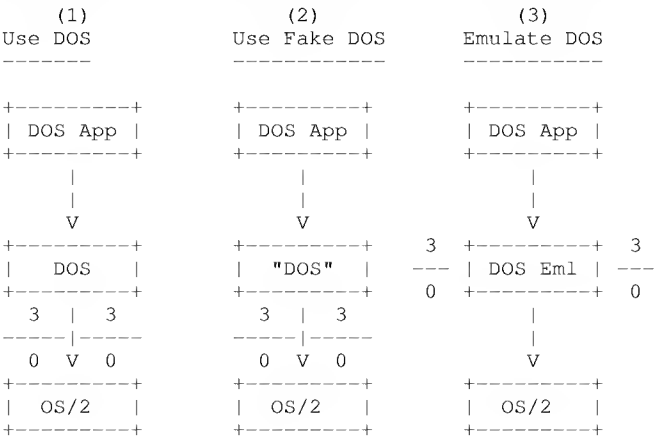
DOS emulation Design

This page is intentionally left blank.

Emulation Strategies

DOS Emulation Strategies

The goal of the DOS emulation component is to find the most compatible and easiest method of DOS emulation. A few of the approaches are not true emulation, but actually run "DOS" or a form of it in a VDM. The following figure diagrams the different approaches.



DOS Emulation Possibilities

The figure shows three possible schemes for supporting the DOS kernel APIs for MVDM. The third scheme is the preferred solution (see following for rationale):

- (1)
- Load real DOS BIOS and DOS into the VDM (as in Xenix/386).
 - + Absolute compatibility.
 - DOS uses memory space in VDM.
 - Must access local file system via REDIR interface.
 -

- Hard errors (INT 24h) will be difficult to handle.
- Hard to make utilities (FORMAT, CHKDSK) work.
- Dependent on another product (DOS) and every OS/2 installation will need a DOS license.
- Hard to detect kernel keyboard polling.
- Hard to restrict old block device drivers.

(2)

Load fake DOS of our own making into the VDM.

- + Total control over DOS support.
- Must revise OS/2 if DOS is revised.
- "Fake" DOS uses memory space in VDM, but not as much as (1).

(3)

Use DOS emulation (as in OS/2 today).

- + Total control over DOS support.
- + Smallest (apparent) code change.
- + Largest (possible) free memory for DOS applications.
- + Better performance because more code runs in protected mode.
- Must revise OS/2 if DOS is revised.
- Harder to maintain DOS compatibility.

The last solution is basically what is done in existing OS/2 systems. There is a large body of existing code that runs in real mode consuming VDM address space that will be converted to run in protected mode. Half of this code (semaphores, interrupt manger, scheduler, etc.) is "dual-mode" to support device drivers. Since device drivers will always run in protected mode, this code can easily be moved out of the VDM address space as protected mode code. The other half of the code is DOS specific support code (like exec, memory alloc, pdb management, etc.) that runs only in real mode. The conversion of this code will be more time consuming.

There are a few requirements for MVDM support:

1. Must be as DOS compatibility as possible.
2. VDM must not negatively affect other VDM or protected mode processes (trashing global OS/2 data structures, etc.).
3. VDM must provide the largest amount free memory for DOS applications.

If the DOS real mode (V86) code is left in the VDM address space and left running in V86 mode, two of the above requirements are not meet. What is even worse is if the VDM's PTDA (per-task-data-area) is left in the VDM address, the system could crash because all PTDA's are iterated for some kernel operations. So the goal is too move and convert all the real mode DOS support code to protected mode without affecting compatibility.

System Services

This page is intentionally left blank.

INT 20h

This service forwards the request to the INT 21h Function 00h service.

INT 21h

All of the INT 21h services that are provided in IBM DOS 4.00 are also provided in this version of the 3XBOX. The internal behavior and error processing behavior of some of the functions may be different. Where such changes are significant, they are listed here:

- Function 00h (ABORT)

If the CS register does not reference the current PDB, the VDM is terminated. In IBM DOS 4.00, the effect of such a call was undefined.

- FCB Functions

Although OS/2 only provides file I/O access externally through file handles, internally, it supports these handles through the System File Table (SFT). In earlier versions of OS/2, FCB I/O was performed by bypassing the file handle processing and manipulating the SFT entries directly. To facilitate this, a special set of SFT entries were reserved for processing FCB's for the 3XBOX.

To take advantage of existing code as much as possible, much of this same method of bypassing file handle I/O will be used. Since OS/2-386 can support more than one 3XBOX, the special SFT entries are allocated dynamically for each VDM, as each VDM is created.

As before, these special SFT entries will be recycled on a least-recently-used (LRU) basis. The recycling will occur on a per-VDM basis, rather than system-wide, since there are multiple VDM's. Similar to earlier implementations, the CONFIG.SYS "FCBS" command will specify the number of these special SFT entries that are to be reserved.

The FCB functions may be called from device drivers during initialization, a functionality that had been unavailable in earlier versions of OS/2.

- Function 38h (International)
- Function 44h (IOCTL)

IOCTL requests which are destined for device drivers within the 3XBOX are processed internally. IOCTL requests which are destined for OS/2 device drivers, however, are treated specially by their respective device drivers. Such requests may contain pointers to data within the 3XBOX. In these cases, it is the responsibility of the OS/2 device driver to perform the necessary translation from V86 virtual addresses into addresses that are meaningful to the device driver.

- Function 5Dh (INTERNAL)
 - Subfunction 0Ah (Set Extended Error Information)

This subfunction allows the calling program to set the register values that are returned from the call to function 59h. This restores functionality that was not present in previous versions of OS/2.

This subfunction function allows Terminate But Stay Resident programs (TSR's) to save and restore the extended error information when they are invoked.

- Function 5F (Name Pipe APIs)
 - Subfunction 5f32 DosQnmPipeInfo
 - Subfunction 5f33 DosQNmpHandState
 - Subfunction 5f34 DosSetNmpHandState
 - Subfunction 5f35 DosPeekNmPipe
 - Subfunction 5f36 DosTransactNmPipe
 - Subfunction 5f37 DosCallNmPipe
 - Subfunction 5f38 DosWaitNmPipe
 - Subfunction 5f39 DosRawReadNmPipe
 - Subfunction 5f3a DosRawWriteNmPipe
- Function 66h - Get/Set Code Page
- Function 67h - Set Handle Count

This function restricts the maximum number of open handles to 254, including the four standard devices.

INT 25h/INT 26h - Absolute Disk Read/Write

The read function operates in the same way as in a IBM DOS 4.00 system. The write function, however, is restricted to removable media only, and reports a hard error on non-removable media.

INT 27h - Terminate But Stay Resident

This function operates in the same manner as in a IBM DOS 4.00 system.

System Callback Procedures

The system callback procedures are "hooks" into DOS that allow application programs to change the default processing action that is taken for certain system events.

These hooks are specified in the virtual mode interrupt vector table as trap service routines. By default, the vectors reference a single IRET instruction.

The vectors used to specify the callback routines are:

- INT 22h - Terminate Address

The DOS kernel stores the terminate address at this vector location.

- INT 23h - Control-C Exit Address

The method used to invoke this callback procedure works the same way as in a IBM DOS 4.00 system.

- INT 24h - Critical Error Handler

Hard errors are normally generated from within the OS/2 kernel. When such errors are detected, the file system checks to see if the requesting task is a 3XBOX. If so, the error indication is returned to the DOS emulation component, still in kernel mode, which determines if the INT 24h vector has been changed. If it has not, the normal OS/2 hard error daemon is called to display the hard error information and to prompt for a reply. If it has been changed, error processing proceeds as at would at this point in a IBM DOS 4.00 system, to the newly specified critical error handler.

- INT 28h - Idle Loop

The method used in calling the INT 28h callback routine is similar to that of a IBM DOS 4.00 system, but this method takes into account the fact that the 3XBOX is running in a multi-tasking environment.

VDM Termination

When a VDM is terminated, the DOS emulation component closes all handles that have been assigned to all PDB processes within the VDM, and all resources associated with FCB's that have been opened are also closed.

Since the VDM may have been terminated due to a catastrophic error within the VDM, no data within the VDM is relied upon to close resources and clean up, but the files and OS/2 devices opened by this VDM are still closed.

Standard DOS Device Drivers

As in DOS, DOS emulation includes DOS device drivers CON, AUX, and PRN. While DOS packages these device drivers in IO.SYS (aka IBMBIO.COM), DOS emulation links these into the DOS emulation kernel (DOSKRNL) to avoid the need for another file. Unlike DOS, the AUX and PRN drivers do not include support for COM1, COM2, LPT1, LPT2, and LPT3. These latter devices are supported by the OS/2 asynch (COM0n.SYS) and printer (PRINT0n.SYS) device drivers.

This approach allows the CON, AUX, and PRN drivers to behave in a very compatible fashion - they issue ROM BIOS calls (INT 10, 14, and 17, respectively) to get their work done. At the same time, using the OS/2 device drivers for the numbered device allows higher performance than is possible through the ROM BIOS interfaces.

In addition, this split allows a numbered device to be redirected to a remote device on a network, using the underlying OS/2 mechanisms.

Of CON, AUX, and PRN, only PRN redirection is supported. This is achieved by the Printer VDD (VLPT.SYS), which routes INT 17 and direct hardware printing to LPT1, LPT2, or LPT3 by using the OS/2 file system. Please see the section on VLPT for more details.

DPMI API Layer Architecture

This page is intentionally left blank.

Problem Description/Objectives

This component provides the INT 31h DOS Protect Mode Interface (DPMI) for VDMs.

DPMI support must satisfy the following goals:

1. DPMI 0.9 must be fully supported.
2. DPMI API (INT 31H) support must be outside the kernel (VDD).
3. Kernel support for DPMI API layer must have a small well defined interface to allow separate development of the 2 components.
4. Both kernel support and the DPMI API layer must be independently expandable to future versions of DPMI. Kernel support for 1.0 must also be able to support a DPMI API VDD requesting 0.9 level support. Conversely, a DPMI API VDD finding only 0.9 level support available must act as a DPMI 0.9 service provider. DPMI's neat compartmentalization of function make this goal a simple one to realize.
5. No fundamental redesign of the support for DPMI 0.9 should be necessary for upgrading to DPMI 1.0. The design here should only need to be added to, not redone.

DOS API Translation is an extension of DPMI needed by some applications. It is described here in a note only to indicate how this design makes it possible to write a DOS API translation VDD with no additional kernel support.

Since many readers are unfamiliar with DPMI, a brief description of the behavior of DPMI applications will be included here. A few notes on terminology are useful. A DPMI host is the provider of DPMI services for the application. Also, V86 mode and real mode will be used interchangeably. From the applications point of view, it makes no difference whether the actual mode is real mode or V86 mode.

Protect mode entry

DPMI applications start in real mode and call an INT 2fh multiplex function to determine if a DPMI host is present. If a host is present, the host returns a "protect mode entry" switch address.

The application calls the switch address to enter protect mode for the first time. When an application returns from the protect mode entry call, it is running in protect mode with its segments converted to protect mode selectors. At this point, the application can use any of the INT 31h DPMI services and can run in either protect mode or in V86 mode. This means that a protect mode application can continue to use DOS and BIOS services or can continue to execute its own real mode code.

The notion of protect mode entry is an important one. Many operating systems provide multiple virtual machines. In a single virtual machine, there may also be multiple DPMI tasks. Each call to the protect mode entry address defines a new DPMI task.

Interrupts and exceptions

DPMI hosts provide protect mode interrupt and exception vector tables. Clients can hook into either table in protect mode.

Hardware interrupts are always reflected first to protect mode hooks. If the application protect mode hooks pass the interrupt to the end of the protect mode interrupt chain, the interrupt is then reflected in V86 mode to the V86 interrupt chain.

Exceptions in real mode are passed to the real mode interrupt handler. Exceptions in protect mode are routed to the entry in the protect mode exception table. If application handlers chain to the end of the exception chain, the exception is reflected next to the protect mode interrupt chain. If this too reaches the end of the chain, the behavior depends on the exception. Exceptions 0-5 and 7 are reflected to V86 mode. The others terminate the virtual machine if not handled in protect mode.

Software interrupts are reflected to the chain in the mode they occur in. If the end of the protect mode chain is reached, it reflects to the V86 chain. Interrupts 23h, 24h, and 1Ch are always reflected to protect mode no matter what mode they occur in.

Hardware interrupts, exceptions, and interrupts 23h, 24h and 1Ch are reflected on a host provided stack in protect mode.

Termination

DPMI applications terminate by doing an INT 21h, service 4Ch in protect mode. Only the last DPMI task to use the entry point (in a virtual machine) can terminate.

DPMI services (See DPMI 0.9 specification for details)

DPMI provides 6 main classes of services:

- Linear memory allocation
- LDT code and data descriptor allocation
- Cross mode calls and switches between protect and V86 mode
- Interrupt and Exception hooking
- Interrupt flag manipulation
- Debug Watchpoint allocation

The following is a complete list of DPMI 0.9 services.

- LDT Descriptor Management
 - Allocate descriptor

- Free descriptor
 - Segment to descriptor
 - Get selector increment value
 - Get segment base
 - Set segment base
 - Set descriptor access rights
 - Create CS alias
 - Get descriptor
 - Set descriptor
 - Allocate specific descriptor
- DOS Memory Management
 - Allocate DOS memory and selector set
 - Free DOS memory and selector set
 - Resize DOS memory and selector set
- Interrupt Hooking
 - Get real mode vector
 - Set real mode vector
 - Get protect mode exception vector
 - Set protect mode exception vector
 - Get protect mode interrupt vector
 - Set protect mode interrupt vector
- Translation (protect/V86 control transfer)
 - Simulate real mode interrupt
 - Far call to real mode
 - Far call to real mode with IRET frame
 - Real mode callback (call protect mode from real mode)
 - Free real mode callback
 - State save/restore for each mode
 - Raw mode switching
- Get Version
- Memory Management
 - Get memory information
 - Allocate
 - Free
 - Resize
- Page Locking
 - Lock
 - Unlock
 - Mark as pageable
 - Relock
 - Get page size
- Demand Page Tuning
 - Mark for demand paging
 - Discard page
- Physical Address Mapping
- Virtual Interrupt State
 - Get and disable IF
 - Get and enable IF
 - Get IF
- Get Vendor Specific Entry Point

- Debug Registers
 - Allocate and set watchpoint
 - Free watchpoint
 - Get watchpoint state
 - Reset watchpoint state

Solutions/Justification

DPMI support is divided into 3 components:

- DPMI API VDD for servicing application INT 31H DPMI requests.
- Kernel support for the DPMI VDD and the VPIC.
- VPIC protect mode hardware interrupt routing.

DPMI is service request driven. An application makes an INT 31H service request. The DPMI VDD handles the request, calling the kernel for basic services like allocating memory.

The design below is for DPMI Version 0.9. The design is easily expandable to DPMI 1.0, however. VPIC will be unchanged when upgrading; its design allows the kernel to make adjustments for the DPMI version. Each of the other 2 components can be written for either DPMI 1.0 or DPMI 0.9 and still work together.

The DPMI 1.0 specification has not been released and there is no need to go into its details here beyond saying that DPMI 1.0 is a superset of DPMI 0.9. The only exception is that error codes have been added to 0.9 calls. The 0.9 implementation described here will include the DPMI 1.0 error codes so that no changes will be necessary to those services when upgrading to DPMI 1.0.

New services can easily be plugged into the DPMI VDD service router (described below) making addition of DPMI 1.0 services simple. New kernel services for use by the DPMI VDD can easily be added to the kernel services available to the DPMI VDD. DPMI 1.0 kernel support, while not discussed here, requires only a small set of new kernel services. Most of the kernel changes necessary for DPMI 1.0 are visible to the application, but invisible to the DPMI VDD. Current VDH services are powerful enough to implement almost all of the proposed new DPMI features in the DPMI API VDD.

In cases where extra work is minimal, DPMI 1.0 functionality has been incorporated in the design for 0.9. The general rule followed is that if a kernel service needs minor enhanced capabilities in DPMI 1.0, then that support is added here.

The following is an overview of the DPMI design. Details can be found in the design section below.

DPMI API Layer communication with the kernel

The API VDD registers with the kernel through a VDH call at system initialization and reports the version of DPMI it supports. If a VDD requested DPMI 0.9 support, a DPMI 1.0 ready kernel would make only a small adjustment in how it handles task creation.

Kernel services that may be useful to VDD's other than the DPMI API Layer are exported as VDH services. Services that should have use restricted only to the DPMI API Layer are made available through structures exchanged when the DPMI API VDD registers.

Kernel functions imported by the DPMI API Layer that are not VDH services:

- DBGWPAAllocVDM
- DBGWPFreeVDM
- DBGDR6QueryVDM
- SELCreateVdmLDT
- SELDestroyVdmLDT
- SELAllocVdm
- SELFreeVdm
- SELGetVdmInfo
- PGMove
- PGGetAccess
- PGAllocSizeQuery
- VMAllocMem
- VMSetMem

- VMFreeMem
- em86DPMITaskSwitch
- SEL_VDM_LDT_INCR value
- CpuType value
- MaxAppLinAddr value
- V86StackSize value

Functions exported to kernel by the DPMI API Layer

- dpmCreateDPMITask
- dpmEndDPMITask
- dpmRouter

The functions are exported by exchanging structures containing pointers. This allows V8086 emulator to mediate between DPMI API Layer and the actual kernel services where this is more convenient. This interface is expandable for supporting DPMI versions beyond 0.9.

This is a list of VDH services added for the DPMI API Layer.

- VDHRegisterDPMI
- VDHGetSelBase
- VDHGetVPMExcept
- VDHSetVPMExcept
- VDHChangeVPMIF
- VDHChangeVPMEM
- VDHRaiseException
- VDHReadUbuf
- VDHWriteUbuf
- VDHCheckPagePerm
- VDHSwitchToVPM
- VDHSwitchToV86
- VDHCheckVPMIntVector
- VDHGetVPMIntVector
- VDHSetVPMIntVector
- VDHArmVPMBPHook
- VDHEndUseVPMStack
- VDHBeginUseVPMStack
- VDHStartHWInt

DPMI API Layer

The API layer provides the following:

- INT 31h routing to service routines.
- Service routines for all 0.9 DPMI services.
- Per task and per VDM client resource tracking.
- Protect mode entry API layer per task initialization.
- Termination API layer clean up.
- Notification to kernel when DPMI task switches occur.
- Entry point exchange with the kernel

Kernel support

- V8086 emulator DPMI support
 - DPMI task entry, termination, mode tracking, control

- New VDH service support
- Get/set support for protect mode handler interrupt and exception handlers
- Interrupt and exception reflection to protect mode
- Protect mode interrupt flag virtualization
- HW interrupt support for the VPIC
- Services to read/write user space with exception handling
- Kernel and VDH service changes for exception handling when accessing user address space
- Debug watchpoint management
- New memory management support for DPML

VPIC support

- Report PIC base changes in VDM
 - Route hardware interrupts to protect mode
-

Faults in ring 0 due to user space access

All VDM addresses linear below 1Mb + 64K can be accessed without visible faults by ring 0 code. Any faults that occur are handled by the Page Manager, VDDs or the VDM Manager. This meant that there was no need to recover from faults at ring 0 when VDM applications only ran in V86 mode.

DPML protect mode apps, however, do have addresses in their address space that can cause visible exceptions at ring 0.

Kinds of exceptions that can occur:

- Loading an invalid selector or not present selector
- Selector limit violation
- Write using a read only selector
- Accessing an invalid page
- Writing to a read only page
- Accessing a supervisor page

The last two do not cause exceptions at ring 0 on a 386. They must be detected and the exception simulated.

Most VDD's are not affected because they never execute while the client is in protect mode.

VDD owners must review their VDD's to see if they are affected. VDD's are affected only if both of the following are true:

- The VDD runs while the client is in protect mode.
- The VDD accesses the client address space above 1Mb+64K or using client selectors while the client is in protect mode. This can happen indirectly if a VDH service is called that manipulates the client's protect mode stack.

How VDD's can gain control while an app is in protect mode:

- IO port trapping
- STI hook
- Call from device driver (or NPX) while application is in protect mode
- Timeout hook (interrupt time)
- VDM Context hook when VDM is in protect mode
- Invalid page fault handler (faulting page is below 1Mb+64K).
- Hitting a PROTECT MODE breakpoint (not a V86 breakpoint)
- Return hooks only if armed in protect mode

Hooks that will not occur in protect mode

- Interrupt hooks

- V86 breakpoints
- Arm return hooks done in V86 mode

If an exception happens at ring 0, the ring 0 code that caused the exception must exit to allow the application exception handler to exit when ring3 is returned to. If the application handler takes care of the exception condition and does not change the application's CS:EIP, the application instruction that trapped to ring 0 will be repeated.

Services will be made available to handle exceptions while reading or writing user buffers. VDD's, however, must be prepared to unwind back to the application when the copy function indicates it has raised an exception. The same is true when using VDH services that access the client's stack when the client is in protect mode.

All VDD's must be checked to see if they could fault at ring 0.

DPMI API Layer Design

This page is intentionally left blank.

Design Overview

A thorough description of the DPMI API Layer will encompass most of this section. Before proceeding to the API layer, brief descriptions of kernel support and VPIC support will be given.

DPMI API Layer communication with the kernel

The DPMI API Layer will have a small well defined interface with the kernel. At system initialization time, the DPMI API Layer uses VDHRegisterDPMI to exchange entry points. This method is used to avoid creating a large set of new VDH services meant specifically for the DPMI API Layer. Keeping the interfaces private also discourages use by other VDDs. The entry points exchanged are listed in the architecture section above. During registration, the DPMI API Layer also reports what version of DPMI it supports. If the VDD supports only 0.9, The kernel refrains from doing some per DPMI task work. Since this is not a DPMI 1.0 design, no further details will be given here.

In V86 mode, all the client address space can be accessed by the kernel without the possibility of a visible fault. In protect mode, on the other hand, accessing the application address space can cause a faults at ring 0. A discussion of this problem can be found at the end of the architecture section.

Client Register Frame (CRF)

The CRF is a structure on the ring 0 stack that contains all of a VDM's registers. The CRF also contains space to save the segment registers of the mode other than the one currently running. This is saved above the start of the ring 0 stack and so is always available at a constant offset from the standard CRF. The registers for the mode not currently running are called the Alternate Register Set. For example, if the application is in V86 mode, the alternate register set contains the selectors that will be used if it switches to protect mode.

KERNEL SUPPORT

V8086 emulator DPMI support

The following is a list of the tasks V8086 emulator performs for DOS protect mode support.

- DPMI task entry, termination, mode tracking, control V8086 emulator exports an instance data pointer to the MVDM DPMI task structure described below. When task switches occur, V8086 emulator updates this pointer. V8086 emulator also provides VDHRegisterDPMI to exchange entry points with the DPMI API Layer.

V8086 emulator provides a protect mode switch entry point in response to an application's INT 2fh multiplex request. When the application calls the protect mode entry to switch to protect mode, V8086 emulator sets up tables for interrupt and exceptions reflection and initializes IDT and LDT. It also notifies the DPMI API Layer, which sets up the initial selectors. If the DPMI API Layer fails the creation call, V8086 emulator cleans up and returns the error to the application.

At task termination, V8086 emulator cleans up and notifies the DPMI API Layer.

V8086 emulator responds to an application INT 2fH request to determine its current mode. To do this it must record the mode each time ring 0 is entered in a VDM since the INT 2fH hook is called after switching to V86 mode to prepare for interrupt simulation to V86 mode. In other words, the application is always in V86 mode by the time the kernel pre-reflection hook is reached. It is the mode of the application when the kernel was entered that is the value needed for return.

V8086 emulator also responds to an application INT 2fH request to give up its time slice. See the VPOL section of the workbook. This function is currently done in the DOSKRNL in V86code. Moving it to the code that handles other INT 2fH multiplex functions in the kernel avoids unnecessarily reflecting down the often long INT 2fH chain when the request is actually one the kernel handles.

V8086 emulator provides a VDH service to switch modes (protect or V86). This service switches the alternate and regular register sets in the CRF. It also handles the virtual IF flag as discussed below.

Another entry point accepts DPMI task change reports from the DPMI API layer. This task change report is used to ensure the correct task has per task allocations assigned to it and to ensure only the last DPMI task that performed entry is allowed to terminate. The report has a more important function in DPMI 1.0.

V8086 emulator exports instance data flags (flVDMStatus) that indicates whether:

- The VDM is running a DPMI application
 - Application is a 32 bit DPMI application
 - Application was in protect mode when ring 0 was entered
 - Current virtual IF interrupt flag state
 - Whether INT 1CH is hooked in protect mode
 - Whether DPMI VDD allows DPMI in this VDM
- New VDH service support

See the headers for the services listed in the architecture section above.

- Get/set support for protect mode handler interrupt and exception handlers

Protect mode applications get and set vectors as in DOS and jump down the chain if they choose to. V8086 emulator maintains tables of protect mode interrupt handlers and exception handlers. These services change or return the current value from these tables.

If the timer tick interrupt, 1Ch, is being set, a bit in flVDMStatus is set to indicate the application is hooking timer ticks in protect mode. If the interrupt being set is one of the hardware interrupts, a bit is set in a bit flag to indicate that the hardware interrupt is hooked. This variable is exported to VDD's in the same way that flVDMStatus is. These bits can be used by vtimer to avoid unnecessary timer tick reflection. They are also used by the VPIC for hardware interrupt simulation. See below for more details.

- Interrupt and Exception reflection to protect mode

Reflection is done by changing the CS:EIP in the applications CRF to reflect the handler to be jumped to. Hardware and software interrupts and exceptions have different rules for whether interrupts are disabled, whether the locked protect mode stack is used, and whether protect mode handlers are invoked when the action is initiated in V86 mode. Exception reflection also has register values placed on the stack for use by the application exception handler. See the DPMI specification and the V8086 emulator section of the workbook.

VDHRaiseException is a new service to reflect an exception to the application. This service arranges for a call to the relevant exception handler when ring 0 is exited. VNPX, for instance, must reflect exception 7h with this service rather than simply by reflecting an interrupt. In V86 mode, this could simply be reflected as an interrupt. In protect mode, a different hook is called for exceptions than for interrupts.

V8086 emulator must record CS:EIP whenever a protect mode application traps into ring 0 before it changes the client EIP. This value is needed if an exception occurs when accessing user addresses at ring 0. It is placed in the stack frame passed to application's exception handlers. This is the instruction that will be reexecuted after the handler corrects the exception condition to restart the trap into the kernel.

- Protect mode interrupt flag virtualization

V8086 emulator should virtualize the IF flag while in protect mode. In V86 mode, IOPL is usually 3 and applications directly change IF without trapping. IF flag virtualization is not done while in V86 mode because IOPL needs to be 3 to cut down on overhead due to trapping STI, CLI, and IRET. In protect mode, IOPL cannot be 3 or else no port protection is possible. Therefore there is no reason not to virtualize the IF flag. This prevents VDM's from blocking out real interrupts when running in protect mode.

To do this, V8086 emulator maintains a virtual IF bit in the flVDMStatus variable. When it traps STI or CLI in protect mode, it modifies this bit rather than changing EFLAGS as it does in V86 mode.

To determine if interrupts are allowed in a VDM that has a DPMI application running, the real VDM IF bit in the CRF is checked first. If interrupts are disabled here, then they are disabled. Otherwise, the virtual IF flag indicates whether interrupts are disabled.

When switching modes, V8086 emulator must switch IOPL and reset the EFLAGS IF and virtual IF appropriately. V86 mode can also use the virtual IF flag when running at less than IOPL 3 to trap STI when STI hooks are waiting.

VDHChangeVPMIF enables or disables interrupts in VDM. When the DPMI API Layer asks to turn interrupts back on, V8086 emulator notifies any VDD's that are waiting for interrupts to be enabled.

- HW interrupt support for the VPIC

V8086 emulator exports a VDH service to accept notification from VPIC when it starts and stops hardware interrupt reflection. This report is ignored for DPMI 0.9 support, but becomes important in DPMI 1.0 where different interrupt tables may be used to reflect hardware interrupts than the current table. Including it now makes it so VPIC does not need to change when DPMI is upgraded.

V8086 emulator also tracks which hardware interrupts are hooked. V8086 emulator will export a pointer to an array containing the following information about hardware interrupts.

- number of pics
- pic base
- 8 bits for which interrupts are hooked

The last 2 entries are repeated for as many PICs as are on the system. VPIC allocates and initializes the buffer at creation time in each VDM, setting the pointer V8086 emulator exports.

Any VDD can use this structure to determine if a particular IRQ is hooked. The timer VDD can use this to avoid delivering timer ticks when the timer tick interrupts are not hooked in either protect mode or in V86 mode.

In DPMI 0.9, just finding out if the related interrupt is hooked provides the information on whether the VPIC needs to reflect an interrupt to protect mode. This is not sufficient in DPMI 1.0 however. This structure provides the VPIC with a reliable way to know which IRQ's are hooked in the task that receives hardware interrupts even if it is not the current task.

When software interrupts are hooked, V8086 emulator refers to this structure to determine if the interrupt is a hardware interrupt. If it is, it sets a bit appropriately in the structure.

VPIC updates the structure and bit maps when the PIC base changes.

- Services to read/write user space with exception handling

V8086 emulator provides a service to copy to or from the user address space while handling exceptions. DPMI apps receive these exceptions as if the exception occurred in ring 3. This is discussed in the notes below and also in the architecture section.

- Kernel and VDH service changes for exception handling when accessing user address space

Services called when the client is in protect mode that manipulate the protect mode client address space must change to handle protect mode user space access exceptions. Services that cannot be called when the client is in protect mode must specify that in their header.

When a service can fault in protect mode, it must return a failure indication to its caller. The caller then cleans up and exits kernel mode so that the exception can be reflected to the VDM. This error return also indicates whether a protect mode exception handler will be called.

If an application exception handler executes, the application instruction that caused entry to ring 0 will be repeated unless the application exception handler changed the CS:EIP.

Here are some of the cases of interest.

- Arming return hooks (stack)
- VDHPushStack, VDHPopStack (stack)
- Instruction decoding (code)
- IO simulation (code, data segment) rep insw, rep outsw

Debug watchpoint management

Coordinates watchpoint use with OS2 protect mode and kernel debugger.

- Allocate and set watchpoint service
parameters
linear address for watchpoint
size of watchpoint (1,2, or 4)
type of watchpoint (execute, write, R/W)
flag - local or global (only local for cruiser).
address for returned watchpoint number
returns SUCCESS/FAILURE
watchpoint number
- Free watchpoint service
parameters
watchpoint number
returns SUCCESS/FAILURE
fails if watchpoint not allocated
- Access virtual B0-B3 bits for allocated watchpoints service.
parameters
Mask of watchpoint bits to get
Mask of watchpoint bits to reset
address for return bits for returned bit values
B0-B3 bits correspond to all allocated watchpoints
returns SUCCESS/FAILURE
fails if watchpoint not allocated
if success, return variable has bits set

Task switching behavior when VDM has allocated a watchpoint:

```
protect mode -> VDM
clear all protect mode global watchpoints
restore VDM's local watchpoints if there are any

VDM -> protect mode
restore protect mode global watchpoints
clear all local watchpoints (current behavior)
```

New memory management support for DPMI

The following kernel services are provided by the VMM:

- Allocate VDM LDT
- Free VDM LDT

- Allocate contiguous set of LDT descriptors (possibly specific set)
- Free descriptor
- Query low 7 pte bits for range of pages
- Query maximum private linear address and ranges of physical memory
- Query maximum linear region and swap space available
- Change linear memory arena size when no conflict
- Other memory management services generally used by the kernel are also used. Services to allocate, free, and set sparse allocations are among those used.

Once descriptors are allocated they are changed directly by the DPMI API Layer. Allocated descriptors must always have nonzero DPL values. Applications set descriptors only through requests to the DPMI API Layer which prevents settings that compromise protection.

VPIC hardware interrupt routing

V8086 emulator exports a pointer to a structure describing the base vector for PICs and whether hardware interrupts are hooked in protect mode.

As described above, VPIC allocates and initializes the structure at creation time and resets it when the PIC base changes. When the PIC base changes, the VPIC uses a kernel service to indicate that hardware interrupts are starting (setting the interrupt table to the one used for hardware interrupts). It then uses a VDH service to determine which interrupts are hooked so that it can fill in the IRQ hooked bitmap for the changed PIC. When finished, it signals that hardware interrupt reflection has ended.

VPIC also reports PIC base vector changes using VDHPutSysValue

To be ready for DPMI 1.0, the VPIC must use the hardware interrupt bitmap to determine if an interrupt is hooked rather than simply querying whether the software interrupt is hooked. It must also report the beginning and end of interrupt simulation when an application is a protect mode application. This allows the kernel to treat hardware interrupts differently, possibly using a different interrupt table than the current one. It then uses other VDH services to switch protect mode stacks and to simulate the interrupt.

VPIC delays reflecting hardware interrupts unless both the CRF EFLAGS IF is set and the virtual IF is set. It uses a VDH service "STI hook" to be notified when interrupts are enabled again if interrupts are disabled.

Data structure overview

A pointer to a structure will be made available to the DPMI API Layer by V8086 emulator containing the following:

- Pointer to previous DPMI task's structure
- Pointer to DPMI API Layer DPMI task data
- Pointer to V8086 emulator DPMI task data
- Task V86 stack segment
- Task LDT address
- Task V86 PSP, ENV, CS, DS, SS
- Task initial protect mode CS, SS, DS

This structure will be referred to as the MVDM DPMI task structure to distinguish it from the data area used exclusively by either DPMI API Layer or V8086 emulator. The 2 data areas pointed to by this structure allow complete separation between data for the DPMI API Layer and V8086 emulator.

The address of this master DPMI task structure serves as the task's ID. V8086 emulator ensures the pointer to the structure is that of the current DPMI task in a VDM when there are multiple tasks.

The remainder of this section is discusses the DPMI API Layer design.

DPMI API LAYER DESIGN

The DPMI API Layer will maintain some data that is VDM instance data and other data that is per DPMI task data. The per DPMI data can be placed either in the DPMI API Layer task data block or in VDM instance data with an associated task ID indicating which task the data is for. This is an implementation detail. The key instance and per task data are listed next. Instance data is available from any DPMI task in a VDM. Task data is available only for the current task.

Per DPMI task data

- Memory allocations
- Descriptors allocated that can be changed by the application
- List header for protect mode to V86 mode calls
- List header for V86 mode to protect mode calls
- List header for DOS memory/selector allocation records
- List header for pending DOS memory records
- List of allocated real mode callback breakpoints
- Handle for protect mode call return hook
- Handle for real mode callback return hook
- Handle for DOS call return hook
- Breakpoints for raw mode switch

VDM instance data

- V86 Segment to selector mapping data
- Debug watchpoint data
- Current LDT address
- Current task ID (pointer to V8086 emulatorDPMI task structure)
- Breakpoints for state saving
- List header for per Task data for terminated tasks
- Count of DPMI tasks

DPMI Task entry initialization

When V8086 emulator traps an application call to the protect mode entry switch, it calls the DPMI API layer. The MVDM DPMI task data described above is available through an exported pointer. The API layer allocates a buffer for its per DPMI task data and places a pointer to it in the MVDM DPMI task structure and initializes its per task data. V8086 emulator maintains the list of tasks and switches the current task. This provides the API layer with a pointer to the relevant task during all service calls.

The DPMI API Layer allocates and initializes selectors for the V86 segments in the MVDM DPMI task structure and places the protect mode selector values in that structure. DPMI allows clients to change only some descriptors allocated to them. The initial CS, DS, and SS are among those the client is allowed to modify. DPMI API Layer notes that they are changeable when it allocates them.

Return hooks are allocated for trapping returns from calls between V86 and protect mode. Breakpoints are also allocated for trapping raw mode switches, and state save calls. These functions will be described below.

DPMI Task termination

V8086 emulator traps an application INT 21h, AX=4Ch termination call in protect mode, and calls the DPMI API layer to allow cleanup. Per task resources that are not cleaned up in Win 3.0, also will not be freed in OS/2 for compatibility reasons.

Per DPMI data blocks will be placed on a terminated list and clean up will be completed when all DPMI tasks exit.

DPMI INT 31h routing

V8086 emulator calls the DPMI API Layer when an INT 31h is done by an application in protect mode. The call is made with the applications state still set to protect mode.

DPMI services are divided into the categories listed above. The router uses a 2 dimensional call table to route to the proper service.

DPMI Service routines

All the services do input parameter checking to validate requests and to enforce restrictions mandated by the DPMI spec.

- LDT Descriptor Management

V8086 emulator arranges for allocation of a tasks LDT. In DPMI 0.9 all tasks in a VDM share the same LDT. LDT Descriptors are allocated through a call to the Selector Manager. Descriptors are allocated with the DPL field set to a nonzero value. This field must never be set to 0 since the Selector Manager is free to use 0 values to indicate the descriptor is free.

Once allocated, descriptors are directly changed by the DPMI API Layer using the LDT base address in the MVDM perDPMI task area. Applications modify descriptors only through a DPMI service calls.

Three types of descriptors must be kept track of:

1. Per DPMI task descriptors that the client can modify
2. V86 segment to selector mappings with descriptors that cannot be modified
3. Per DPMI task DOS descriptors that the client cannot modify

DOS selector tracking is discussed in the design section for DOS memory allocation.

V86 segment to selector tracking will be done using a hash table in the VDM instance data. This data is per VDM, not per task. Each record will include the V86 segment (the key) and the corresponding selector. These selectors will not appear on the list of modifiable selectors and, therefore, are not modifiable by the application.

Modifiable descriptors will be indicated by setting a bit in a bit map. The bitmap is divided into blocks and placed in a hash table. This allows quick access while cutting down on memory use. Memory allocations are owned by DPMI tasks and so the hash table is in the per DPMI area.

- Allocate descriptors

A set of descriptors are obtained from the Selector Manager. The descriptors are initialized and the selector bit map is marked to indicate the client can change these descriptors. When necessary, the bit map is grown to handle new selectors.

- Free descriptor

The call fails if the descriptor is not one of the descriptors permitted to change. The Selector Manager is called to free the descriptor, the bit in the bitmap for the descriptor is cleared, and the count for the block is decremented.

- Segment to descriptor

The segment to descriptor map is checked to see if the selector has already been allocated. If it has been allocated, the selector is returned.

If it has not been allocated, a selector is allocated and initialized to point to the V86 segment and the relation is added to the segment to selector mapping.

This mapping is in VDM instance data and is cleaned up only when all DPMI tasks in the VDM have terminated. The DPMI spec suggests that these selectors be used only for a few important low memory addresses (i.e. the video buffer).

- Get selector increment value

A value is returned that indicates the offset to add to a selector to reach the next one in a set. This is up to the Selector Manager, but contiguous descriptors will be allocated.

- Get segment base
Set segment base
Set descriptor access rights
Get descriptor
Set descriptor

The descriptor is directly changed for these services if the changes are allowed by the DPMI spec. Only ring 3 code and data descriptors are allowed. One important restriction is that descriptors will never be allowed to point to addresses above the private arena. This restriction is permitted in the DPMI spec to protect OS addresses.

- Create CS alias

A new selector is allocated from the Selector Manager. The old descriptor is copied and then set to be a data descriptor. The descriptor bitmap is modified to indicate the descriptor can be changed.

- Allocate specific descriptor

This is done in the same way as the first descriptor allocation except that a specific set of descriptors is requested from the Selector Manager. The first 16 selectors are reserved exclusively for specific selector requests. Any other specific descriptors can be requested as well. The descriptor is checked for availability (checking the DPL bits) before calling the Selector Manager.

- DOS Memory Management

- Allocate DOS memory and selector set

This service allocates DOS memory along with a set of descriptors to cover the allocation. For 32 bit clients, a single descriptor is set to cover the entire allocated region. For 16 bit clients, this descriptor is followed by descriptors to cover the rest of the region in 64K chunks. This allows 16 bit applications to refer either through a single large segment or through tiled selectors.

An actual V86 DOS call is used to allocate memory from the DOS arena. This means that after initial setup is done, kernel mode needs to be exited to call into DOS, and then the return from DOS has to be trapped to finish the service.

Before reflecting to V86 mode to call DOS, the Selector Manager is called to get enough descriptors to cover the DOS allocation in 64K segments. An allocation record is set up and placed on the pending allocation list in instance data. This record stores:

- Type of request (Allocate, Resize, Free)
- First Allocated selector in the set of selectors
- Allocation size
- SS:ESP to identify request
- DPMI task ID of caller (pointer to task data serves as ID)
- Selector used in call
- Requested size (for resize or free)

Descriptors are allocated before reflecting to V86 mode to avoid allocating DOS memory when descriptors are not available.

A DOS call to allocate memory is simulated to V86 mode to call DOS and the return from DOS is trapped (as in the translation interrupt service described below).

When the return from DOS is hooked, the allocation record is removed from the pending list (SS:ESP defines which record to remove). If the DPMI task has changed, the kernel is called to set the current task back to the one that made the call. If the call succeeded, the DPMI API Layer sets the DOS Memory allocation record to show the region was allocated, fills in the descriptors appropriately, and fills in the client's registers with the appropriate return values. The DOS allocation record is placed on the allocation list in the task's data area.

If the DOS call fails, the selectors are freed using a Selector Manager call. The DOS Memory allocation record is freed and an error return to the client is set up. This call can occur recursively.

- Free DOS memory and selector set

First, the allocated list is searched to make sure the region being freed is allocated (search for linear address). The allocation record is moved to the pending list with the request marked as "free." If a different selector than the one

allocated is passed in, the descriptors are checked to ensure the first has the same base and limit and that any descriptors that will be accessed were allocated by the application.

A switch is then made to V86 mode and the INT 21h is simulated as above with the return trapped.

When the return is trapped, return values are set up. If the call succeeded, the selectors that were allocated are freed. If a selector other than the allocated selector was passed in in the free call, that selector set is freed as well.

- Resize DOS memory and selector set

Resize is done like the original allocation so the design will just be sketched. The allocation record is moved to the pending list. The desired size is listed in the allocation record and new selectors are allocated if the size increases and new selectors are needed. Descriptors are allocated before reflection so the call can be failed before allocating DOS memory in V86 mode if they are not available. The DOS call is then done as above.

When the hook regains control, if the call failed, the new selectors are freed. If it succeeded, new descriptors are set up if they were needed or descriptors are freed if the resize made some unnecessary. The return values for the client are set up and the allocation record is returned to the allocated list with its new size noted.

- Interrupt Hooking

- Get real mode vector
Set real mode vector

Changes to the V86 IVT are done directly to the IVT in the VDM address space.

- Get protect mode exception vector
Set protect mode exception vector
Get protect mode interrupt vector
Set protect mode interrupt vector

V8086 emulator maintains a table of the current handler for each protect mode hook and exception. These services are implemented by calling V8086 emulator to get the current value from this table or to set a new value in the table.

- Translation (protect/V86 control transfer)

These services provide provide cross mode calls, state saving, and raw mode switching.

- Simulate real mode interrupt
Far call to real mode
Far call to real mode with IRET frame

All of these calls pass a buffer containing register values to be used in V86 mode. The latter two also transfer parameters from the callers stack to the V86 stack.

The current register state for both modes is saved, the registers specified in the call are set up and after the call, the V86 register values are copied back to the client's buffer and the original saved registers are restored. This requires state saving, handling exceptions when user buffers are accessed, calling across modes and trapping the return after the call.

The first step is to read in the user buffer using a VDH service to handle exceptions. If an exception occurs while getting the buffer, the service returns immediately. The exception is reflected by V8086 emulator. If the exception condition is corrected, the INT 31H will be repeated, but this time the user buffer will be accessible.

For the services that allow passing arguments on the stack, the arguments from the stack are copied to a buffer in the same way.

It is important that transfers from the user space are done into a buffers that will not cause faults. This simplifies cleanup if an exception occurs which must be reflected to the application.

A V86 call record is allocated and the current CRF including the alternate register set for V86 mode is copied into the V86 call record so that it can be restored after the call. The current DPMI handle, the selector and offset of the client's V86 register buffer, the current V86 SS:SP if a stack switch is occurring, and a breakpoint ID are also stored in the record. The call record is added to the per DPMI V86 call list. A return hook is allocated and its data portion is set to contain the DPI task ID and a breakpoint ID (the address of the data portion of the breakpoint).

The mode is then switched to V86 mode to prepare for the call. The registers are set according to the callers request buffer except for CS:IP. The reserved V86 stack is set up if the user did not specify a stack. Arguments are copied to the V86 stack if necessary using a VDH service (no faults possible here because the stack is in V86 space). Another VDH service is then used to set up the return hook trap and another VDH service is used to set up the call or interrupt

into V86 mode.

When the return hook trap occurs, the ID in the breakpoint data area is used to find the correct record on the V86 call list.

If the V86 record's DPMI task ID is different from the current ID, V8086 emulator is called to switch tasks. Calls to and from V86 mode can happen recursively. When a V86 call returns, the DPMI task that caused the call must be restored as the current task. This controls what task owns allocations that are made and whether the task is allowed to terminate.

The client's V86 register buffer is then set with the current V86 register values using a VDH service. If a fault occurs, as above the service exits as above. The only thing to undo is a switch of the current protect mode task if one was done. If the application handles the exception, the V86 instruction that hit the breakpoint will repeat and the return hook can start over.

The client is then switched back to protect mode, the breakpoint is freed, and the V86 record is taken off the call list.

The next step is to restore the saved register state for both modes. The service is finished at this point and exits signalling success.

As an optimization, a single breakpoint can be used with information to identify the particular call on the client's stack.

- Real mode callback (call protect mode from real mode)

This service allocates a real mode callback breakpoint. When this breakpoint is called, the DPMI API Layer handler arranges a protect mode call.

When the allocation call is made, a breakpoint is allocated. The breakpoint data area has the DPMI task ID and the breakpoint data area address placed in it. A real mode callback record is added to a list in the DPMI task area to note the breakpoint allocation. The selector:offset of both the client protect mode procedure to call and the real mode call structure are placed in the real mode callback record. A protect mode selector pointing to the real mode call structure is set up and is stored in the callback record.

When the callback breakpoint is jumped to by the client, the DPMI API Layer allocates a protect mode call record that points to the real mode callback record. The DPMI API Layer saves registers to restore later for protect mode in the protect mode call record.

A descriptor is allocated to address the real mode stack in protect mode and the selector is placed in the protect mode call record.

If the real mode callback is owned by a task other than the current task, the DPMI task is switched. The old task is saved in the protect mode call record.

At this point, a VDH service is used to switch the client to protect mode. The protect mode DS:(E)SI is set to the real mode SS:(E)SP. The protect mode ES:(E)DI is set to the client's real mode call structure. The protect mode SS:SP is saved in the protect mode call structure and a service is used to switch to the protect mode locked stack (the stack used for protect mode hardware interrupts, exceptions, real mode callbacks, and software interrupts 23h, 24h and 1CH).

A return hook is set using a VDH service to catch the return after the protect mode call. The hook returned to is identified by an identifier placed on the user stack. Another service is used to set the protect mode address to return to when exiting kernel mode.

When the return hook traps the application protect mode return, the DPMI API Layer copies the real mode call structure back into the V86 registers. If the protect mode task was switched to make the real mode callback call, it is reset to the previous task. The mode is switched to V86 mode. The protect mode call structure and the descriptor used to point to the real mode stack are freed. (As an optimization, unused descriptors and structures may be kept on a list for reuse).

- Free real mode callback

If a real mode callback is still waiting to be completed, the callback record is marked to indicate it is no longer active. Freeing the callback record and the breakpoint are done when no outstanding calls are in progress.

- State save/restore for each mode

This service returns a set of addresses, one for V86 mode and one for protect mode that if called by the client, save or restore the current register state for the other mode. This is necessary for applications that do raw mode switching to keep them from overwriting the current saved alternate mode registers.

The allocation part of this service returns the address of 2 breakpoints. These are allocated in instance data when the first DPMI task enters protect mode.

When the breakpoint is called the alternate registers are copied to or from the client's buffer. In protect mode, this is done with a VDH service to handle exceptions. In V86 mode, it is done with wrapping in segments simulated.

- Raw mode switching

This service returns a set of addresses, one for V86 mode and one for protect mode that if called by the client, switch to the other mode. The breakpoints have the DPML task ID in the breakpoint data area.

When the breakpoint is hit, if the ID is different than the current one, V8086 emulator is called to report the switch.

A VDH service is then used to do the requested mode switch. DS, ES, SS, SP, CS, and IP are set as specified in registers when the switch address is jumped to. FS and GS are set to 0.

It is possible that interrupt hooks set by one task will allocate resources when another task is running. This makes determining which task owns a resource impossible. Cleanup for some items may need to be postponed until all DPML tasks exit. Only resources cleaned up by Win 3.0 will be cleaned up.

- Get Version

This simply returns 0.9.

- Memory Management

- Get memory information

This service uses memory management calls to fill in an application buffer with a variety of information about the memory. A VDH service is used to copy the data to user space handling with exception handling.

- Allocate

Memory is allocated using a memory management service. Record of the allocation noting the start address, allocated size, and sparse linear address size are kept in a hash table using the HOBj of the object as its key. The allocation records are kept in the DPML API Layer per DPML task area so that they can be cleaned up when the task terminates.

- Free

The allocation is freed using a memory manager service and the allocation record is freed.

- Resize

Resizing may require moving an allocation. Presently, sparse allocation sizes (arena record sizes) cannot be changed even if there is no conflict in the arena. To avoid repeatedly moving allocations, shrinking will be accomplished by decommitting without moving.

- Shrinking resize requests use memory management services to decommit pages. The allocated linear space is left unchanged, but the allocation record is updated to reflect the committed size.

- Growing resize requests that fit in the allocated sparse space are accomplished by asking the memory manager to commit more pages. The committed size is updated in the allocation record.

- Growing allocations that cannot fit in the allocated linear space must be moved. The order in which the move is done is important to allow graceful recovery when failures occur. Memory services are used to accomplish each of the following steps:

- Allocate new size sparse linear space of uncommitted pages
 - Commit needed pages at the top of the new linear range
 - Move the page table entries from the old to the new range
 - Free the old range
 - Update the allocation record and replace it in the hash table since it will have a new HOBj.

Resizing can be accomplished more efficiently in place given the ability to have the arena size adjusted when this is possible. This will be done if the memory management capability is made available.

- Page Locking

- Lock
Unlock
Mark as pageable
Relock

Page locking services are necessary on systems that deliver interrupts at interrupt time or that use DOS for paging. On a system that simulates interrupts and has its own file system the calls are NOP's. These will simply return a success indication.

- Get page size

Returns the page size (4K).

Demand Page Tuning
Mark for demand paging
Discard page

OS/2 does not need help in determining the current working set of an application. These calls return success without doing anything.

- Physical Address Mapping

In OS/2, we have no way of knowing what addresses are used by device drivers. This makes it dangerous to allow direct access to devices that do not have VDDs. However, we do allow direct access in VDM's.

Current plans are to fail all requests to this service. This is allowed in the DPMI spec so that operating systems can protect devices.

If time allows, a property could be added to allow direct physical mapping by DPMI applications. Memory management supplies a list of all physical memory ranges so that if the service is supported, it will avoid mapping to physical memory.

(Our current VDH services for reserving linear space, mapping, and page fault handling are all restricted to regions below 1 Mb+64K. This currently makes it impossible for a VDD to virtualize hardware with linear address above 1 Mb.)

- Virtual Interrupt State

Get and disable IF
Get and enable IF
Get IF

V8086 emulator will offer a service to change the client's IF flag. This service maintains a virtual interrupt IF bit in the variable `fiVDMState`. It also calls any pending VDD STI hooks when interrupts are enabled.

To determine the current IF state, the DPMI API handler first checks the client's EFLAGS. If interrupts are really off, they are considered off. If they are on, then the virtual IF is examined to determine the current state. This leaves V8086 emulator more flexibility in virtualizing IF.

If the current IF state is different from the state the client is setting, the V8086 emulator virtual IF service is called to change the state.

V8086 emulator calls pending STI hooks when interrupts are enabled.

- Get Vendor Specific Entry Point

Vendors that add extensions look for the name of their extension by hooking INT 31h. If their extension is requested, they IRET without jumping down the INT 31h protect mode chain. Since the DPMI API Layer router is called at the end of the protect mode chain, receiving this API request means that the extension is not available. This is signalled to the client by setting the carry flag to indicate the call failed.

- Debug Registers

Allocate and set watchpoint
Free watchpoint
Get watchpoint state
Reset watchpoint state

The Tasking component manages watchpoints for OS/2 applications, the kernel debugger, and VDM's. Interfaces for allocating and setting, freeing watchpoints and getting the Bx bits for allocated watchpoints will be used by the DPMI API Layer to carry out these services.

The DPMI API Layer keeps track of allocated watchpoints in the per DPMI task area so that it can clean up at termination and uses the tasking watchpoint services to manipulate watchpoints.

Imported Interfaces

The DPMI API Layer uses various kernel and VDH services. See the sources for details. The design section above contains a list of new VDH services and also private kernel services used by the DPMI API Layer.

Notes

This section contains ideas that are being considered in the design. These notes apply to DPMI in general and mostly impact 8086 design, not the API layer. These are rough working ideas, not final design.

Notes for handling user space access from kernel for DPMI API layer, DOS API translation module, VDD's.

REQUIRED NEW EXCEPTION SERVICE

New VDH service to pass exception to VDM.

BOOL VDHENTRY VDHRaiseException(iException, ulErrCode, ulExtraCode);

If the app is in protect mode, this goes to the protect mode exception chain.

If the app is in V86 mode, a 16 bit bitflag is checked to see if reflection goes to the protect mode exception list. If it does not go to protect mode, then it is reflected as a V86 interrupt.

VNPX must use this service. This can also be used if it is discovered that a user trap will cause an exception. (e.g. a bad buffer was passed to a dpmi call).

VDHArmExceptionCallback (hook) - (To be specified)

Call back after exception is simulated to allow continuing an operation that was not completed (where client registers can't be updated to restart properly). The callback can be called with a flag that says the CS:IP was changed, so the operation should not continue. It isn't clear whether this is necessary.

SERVICES TO ALLOW PRECHECKING WITHOUT ACTUALLY FAULTING (appropriate for non-performance critical code with large buffer access)

VDH service to check access permission.

BOOL VDHENTRY VDHCheckPagePerm(ppageStart, cPages, pflPerm)

Checks access permission for ring 3.

*ppageStart + cPages must be in user arena

pflPerm	R/W	-- R/W bit set	S/U	-- S/U bit set	Invalid -- invalid page
---------	-----	----------------	-----	----------------	-------------------------

Input	pflPerm	R/W -- Check R/W bit	Invalid and supervisor pages are always checked for.
-------	---------	----------------------	--

Output	ERROR RETURN	*pflPerm will be updated to reflect the reason for failure	*ppageStart will be updated to the page that the failure occurred on.
--------	--------------	--	---

DPMI has some infrequent calls with huge buffers and buffers with variable limits. This will be used to precheck access for these calls. If access is denied VDHPushException will be used to raise an exception. DPMI provides apps with a similar call, so it is simple to also provide it to VDDs and the DOS api translation module.

New service to get selector information

BOOL VDHENTRY VDHGetSelBase(sel, pladdr)

For protect mode selectors only.

The other selector information is available through 386 instructions.

This usually won't be needed, but dpmi has to do it anyway so we may as well make it available for VDD's that want to access protect mode user space with flat pointers.

SERVICES TO ALLOW COPYING WITH EXCEPTIONS HANDLED AUTOMATICALLY

Callers must be ready to undo whatever they have done and return if the call fails in order to allow the exception to be simulated. Exceptions will not necessarily kill the VDM. An app exception handler may correct the condition and restart the instruction. This means that VDD's must leave the app in a state in which it can restart the instruction. Note: most VDD's will never get trapped to in protect mode since they trap V86 specific V86 events (like a V86 interrupt).

For instructions that can be partially completed, the client registers should be updated to reflect how much has already been done. One example is I/O simulation. If an invalid page fault happens in a rep insw edi could be updated to restart where the fault occurred.

These services are already needed for em86 to push on the user stack, so they can be provided to callers at little extra cost.

VDHPushStack/VDHPopStack (em86).

These services should be updated to use the interfaces below. The caller must be able to look for the error return and exit after cleaning up so the exception can be simulated.

Existing VDH services that manipulate user stack (em86).

Each must either be modified to work in protect mode or else must note that the VDD must check. VDD owners should report what services they need in protect mode so we make sure all those are available.

Services to access from user space (em86).

These services access user space as if the CPL was ring 3.

All VDHAllocPages, VDHReallocPages, or instance data allocations are marked supervisor. Users cannot access them directly. They can, however, attempt to access them indirectly by either passing them as a buffer to ring 0 or by setting the stack to point to supervisor pages and then trapping to ring 0 in a way that leads to ring 0 code manipulating the user stack.

Restrictions on protection:

Restrictions on protection allow for much faster checking without compromising security. Wherever possible, the code that transfers to and from user space must be able to perform the access and catch the exception without wasting time checking permissions first. This cannot be done with supervisor page protection since that access is allowed at ring 0. To avoid this page by page checking the following restrictions will hold. Note that our model already requires that all instance data be below 4Mb.

Restrictions:

- Supervisor bits do not have to be checked on reads.

Applications can claim supervisor pages are their input buffer without harm to security. A VDD that copies the input buffer to an output buffer can ask for a check on the input buffer. Otherwise, the user never sees the contents of the supervisor pages.

- (Not sure about this one) Supervisor bits must only be checked on writes when pages are between 1Mb+64K and 4Mb.

- PRO

On a 486 can do writes without checking page permissions. (The 386 is fouled up for read only and must be checked anyway). This would speed up em86. Apps will not be allowed to have their stacks between 1Mb+64K and 4Mb. That will automatically cause a fault. Once that was checked for, page permission checks could be skipped on a 486 (catching read-only faults).

- VDDs with large buffers can check their location and skip supervisor checks if they are entirely above 4Mb.

- CON

If a large amount of secure data is required for a VDD and it can't fit below 4Mb, the VDD would have to put the data in the system arena.

Allows Broken/malicious programs to write on supervisor pages above 4Mb by passing a buffer to a VDD that writes to user space or by setting its stack on top of supervisor pages. (There probably will not be supervisor pages above 4Mb in the space addressable by apps, below 512 Mb).

- Applications will never have allocations between 1Mb+64K and 4Mb, but can point descriptors there. Any user space access in that region will always fault.

Assumes caller has already checked sel is ring 3 LDT. When it is known the address is in the V86 address space, this should not be called.

BOOL VDHENTRY VDHReadUbuf(pDstBuf, cb, sel, pulOffset, fl)

BOOL VDHENTRY VDHWriteUbuf(pDstBuf, cb, sel, pulOffset, fl)

fl -- controls on what to check for R/W -- Fault if R/W bit set S/U -- Fault if S/U bit set CheckSelWrite -- Fault if descriptor is read-only SellsPresent -- Caller knows descriptor is present SellsWritable -- Caller knows descriptor is writable

*pulOffset is updated with the address of the fault. For selector faults it is unchanged

BOOL VDHENTRY VDHReadULin(pDstBuf, cb, pladdr, pfl)

BOOL VDHENTRY VDHWriteULin(pDstBuf, cb, pladdr, pfl)

fl -- controls on what to check for R/W -- Fault if R/W bit set S/U -- Fault if S/U bit set

*pladdr is updated with the address of the fault.

Callers can:

- Skip checking when information is known.
- Skip checking when access is safe even when a violation (supervisor checking on a read).
- Ask to check write permissions on reads when the same buffer will be written to later. It is usually much easier to unwind when a fault is hit early.

Callers must be able to handle failures:

- Back out actions performed so instruction can restart (possibly updating client registers so the instruction restarts after what has already been done).
- Save data and arrange a callback after the exception handler runs to continue the operation. (Service to be specified by Em86)

VDHArmExceptionCallback (hook) - to be specified

DOS API translation

Protect mode applications issuing DOS or BIOS calls must pass buffers that can be accessed in V86 mode. The DOS API translation relieves the application from having to do this work for DOS calls and some BIOS calls. This lets protect mode applications use protect mode buffers pointed to by protect mode selectors in DOS calls. DOS API translation arranges for any necessary buffer copying. DOS API translation is not part of DPML. It is an extension to DPML included here simply to indicate how it fits into DPML.

Applications detect the presence of a DOS API translator by performing an INT 2fh multiplex passing the name "MS-DOS" as an argument. The translator responds when it sees this name, indicating that translation will be performed. Applications that do not require translation use the INT 31h simulate interrupt function to avoid translation.

The DOS API translation component performs the following functions:

- Provides INT 2fh multiplex presence query function.
- Allocates a 4K V86 buffer to use for translation.
- Hooks INT 21h and some BIOS interrupts at the end of the protect mode chain. This means an application could install its own DOS API translator and the system translator would never be called.

These interrupts are hooked by using the VDH get protect mode interrupts service and then using the set service to place a breakpoint in the vector. If done at creation time, this places the breakpoint at the end of the protect mode chain. DOS API translation can jump to the next handler in the chain by putting the CS:EIP it retrieved into the CRF. Otherwise it can directly switch modes and reflect the interrupt.

- For calls that it translates, copies, breaking call into multiple calls for buffers larger than 4K.
- Uses VDH services to reflect the interrupt to V86 mode and also to trap the IRET when output buffers need to be copied.

Imported Interfaces

The DPML API Layer uses various kernel and VDH services. See the sources for details. The design section above contains a list of new VDH services and also private kernel services used by the DPML API Layer.

Notes on faults when accessing DPMI protect mode user space when at ring 0 in a VDM.

In the past, VDDs only accessed addresses they allocated or addresses below 1Mb + 64K. Any address in V86 space is always valid to read or write from. It either is committed or will be taken care of by default or by a VDD that has claimed it. All pages in V86 space are legal for access by applications and segment registers contain V86 segments, not protect mode selectors. All this made accessing user addresses very simple. Exceptions in the kernel were always handled invisibly and protection was not an issue.

DPMI programs have a great deal of control over their address space. They can create conditions in the user address space above 1Mb+64K that will cause exceptions if an application buffer or stack is accessed. They also can hook exceptions. These exceptions must be simulated even when the faults occur at ring 0. If an application passes a buffer to ring 0 code that would cause an exception if accessed at ring 3, it the exception must be simulated from ring 0. Changing the user stack is another example where exceptions can occur at ring 0.

This presents some difficulties. Exception conditions must be detected or caught and when caught execution must unwind to ring 3 to allow the exception to be simulated.

All of the address that were safe before are still safe to access. VDD's can touch anything below 1Mb+64K and any allocations they make. DOS protect mode programs will not be able to access any VDD allocations or instance data.

Exceptions will occur only when the application is in protect mode and when ring 0 code accesses an application supplied buffer, the application's stack, or the application's code segment. Stack access occur indirectly when VDDs call services that use the stack (e.g. VDHArmReturnHook).

The following conditions must hold for a VDD to cause an exception.

- + Application is in protect mode
- + VDD or kernel accesses application user space above 1Mb+64K

Conditions to consider in deciding if a VDD can be affected:

- + Can the VDD be entered in protect mode?
- + Does the VDD access user space above 1Mb + 64K?

**** We need a list of all VDDs that could have these conditions
**** both met. Let me know if you know whether your VDD is
**** affected.

VDD's that are not affected:

- VEMM: Entered only through hooking int 67h or from hitting V86 mode breakpoints. It can never be entered in protect mode.
- VXMS: Same as VEMM.
- VPOL: (idle detection) Never accesses user space or a user selector.
- VVIDEO: Never accesses user addresses above 1Mb + 64K.

VDD's that are affected:

- VPIC: Can be called when in protect mode and accesses the application stack.
- VNPX: Called while in protect mode. Must use new VDPushException to cause exception 7h.

VNPX was to have to emulate some instructions to make it look like a coprocessor was not present when it really was (errata).
(finitwhen the coprocessor was being hidden for

It currently does not do this (and I think is broken). It will need to take care in reading the code if it emulates instructions.

VDMA Accesses user space.

Are these affected?

VBIOS:
VCMOS:
VCOM:
VDSK:
VFLPY:
VKBD:
VLPT:
VMEM:
VMOUSE:
VTIMER:

How VDD's can gain control while an app is in protect mode:

- + IO port trapping
- + VDM Context hook when VDM is in protect mode
- + Hitting a PROTECT MODE breakpoint (not a V86 breakpoint)
- + Call from device driver (or NPX) while application in protect mode
- + Return hooks only if armed in protect mode
- + STI hook
- + Invalid page fault handler (but should only be touching the faulting page below 1Mb+64K).
- + Timeout hook
- (any others?)

Hooks that will not occur in protect mode

- + Interrupt hooks
- + V86 breakpoints
- + Arm return hooks done in V86 mode
- (any others?)

If an exception happens at ring 0, the ring 0 code that caused the exception must unwind to a state that would allow the instruction that caused the trap to ring 0 to be repeated. On the way out of kernel mode, the exception is simulated to the application. An application handler can then correct the exception condition or change the cs:ip that will be executed after exception handling is completed. An alternative is for the VDD to save information about what it is doing, arrange to be called after the exception is simulated and to continue then.

Services will be made available to make this as painless as possible.

Kinds of exceptions that can occur:

- + Loading an invalid selector or not present selector
- + Selector limit violation
- + Write using a read only selector
- + Accessing an invalid page
- + Writing to a read only page
- + Accessing a supervisor page

The last 2 do not cause exceptions at ring 0 on a 386. They must be detected and the exception simulated.

Protection restrictions:

- + Applications will be allowed to pass read only pages as input buffers (no checking done). If a VDD is going to return the contents of the input buffer in another buffer it will ask to check read only. Otherwise, no check needs to be done.
- + Supervisor pages will only be checked for on writes below 4Mb to save time. All important VDD data can fit below 4Mb. VDHAllocPages will have a flag value added for a request below 4Mb. (VEMM allocations can be stepped on if an app really wants to). (Still thinking about this one - it helps only on 486's since read only always must be checked).
- + Applications will never have allocations between 1Mb+64K and 4Mb, but can point descriptors there. Any user space access in that region will always fault.

Notes for handling user space access from kernel
for DPMI API layer, DOS API translation module, VDD's.

REQUIRED NEW SERVICE

1. New VDH service to pass exception to VDM.

```
BOOL VDHENTRY VDHPushException(iException, ulErrCode, ulExtraCode);
```

If the app is in protect mode, this goes to the protect mode exception chain.

If the app is in V86 mode, a 16 bit bitflag is checked to see if reflection goes to the protect mode exception list. If it doesn't go to protect mode, then it is reflected as a V86 interrupt.

VNPX must use this service. This can also be used if it is discovered that a user trap will cause an exception. (e.g. a bad buffer was passed to a dpmi call).

VDHArmExceptionCallback (hook) - to be specified call back after exception is simulated to allow continuing an operation that was not completed (where client registers can't be updated to restart properly). The callback can be called with a flag that says the cs:ip was changed, so the operation shouldn't continue.

SERVICES TO ALLOW PRECHECKING WITHOUT ACTUALLY FAULTING
(appropriate for non-performance critical code with large buffer access)

2. New VDH service to check access permission.

```
BOOL VDHENTRY VDHCheckPagePerm(ppageStart, cPages, pflPerm)
```

Checks access permission for ring 3.

*ppageStart + cPages must be in user arena

```
pflPerm
  R/W      -- R/W bit set
  S/U      -- S/U bit set
  Invalid  -- invalid page
```

```
Input
  pflPerm
    R/W -- Check R/W bit
    Invalid and supervisor pages are always checked for.
```

```
Output
  ERROR RETURN
  *pflPerm will be updated to reflect the reason for failure
  *ppageStart will be updated to the page that the failure
  occurred on.
```

DPMI has some infrequent calls with huge buffers and buffers with variable limits. This will be used to precheck access for these calls. If access is denied VDHPushException will be used to raise an exception. DPMI provides apps with a similar call, so it is simple to also provide it to VDDs and the DOS api translation module.

3. New service to get selector information

```
BOOL VDHENTRY VDHGetSelBase(sel, pladdr)
```

For protect mode selectors only.

The other selector information is available through 386 instructions.

This usually won't be needed, but dpmi has to do it anyway so we may as well make it available for VDD's that want to access protect mode user space with flat pointers.

SERVICES TO ALLOW COPYING WITH EXCEPTIONS HANDLED AUTOMATICALLY

Callers must be ready to undo whatever they have done and return if the call fails in order to allow the exception to be simulated. Exceptions will not necessarily kill the VDM. An app exception handler may correct the condition and restart the instruction. This means that VDD's must leave the app in a state in which it can restart the instruction. Note: most VDD's will never get trapped to in protect mode since they trap V86 specific V86 events (like a V86 interrupt).

For instructions that can be partially completed, the client registers should be updated to reflect how much has already been done. One example is I/O simulation. If an invalid page fault happens in a rep insw edi could be updated to restart where the fault occurred.

These services are already needed for em86 to push on the user stack, so they can be provided to callers at little extra cost.

4. VDHPushStack/VDHPopStack (em86).

These services should be updated to use the interfaces below. The caller must be able to look for the error return and exit after cleaning up so the exception can be simulated.

5. Existing VDH services that manipulate user stack (em86).

Each must either be modified to work in protect mode or else must note that the VDD must check. VDD owners should report what services they need in protect mode so we make sure all those are available.

4. Services to access from user space (em86).

These services access user space as if the CPL was ring 3.

All VDHAllocPages, VDHReallocPages, or instance data allocations are marked supervisor. Users cannot access them directly. They can, however, attempt to access them indirectly by either passing them as a buffer to ring 0 or by setting the stack to point to supervisor pages and then trapping to ring 0 in a way that leads to ring 0 code manipulating the user stack.

Restrictions on protection:

Restrictions on protection allow for much faster checking without compromising security. Wherever possible, the code that transfers to and from user space must be able to perform the access and catch the exception without wasting time checking permissions first. This cannot be done with supervisor page protection since that access is allowed at ring 0. To avoid this page by page checking the following restrictions will hold. Note that our model already requires that all instance data be below 4Mb.

Restrictions:

- + Supervisor bits do not have to be checked on reads.

Applications can claim supervisor pages are their input buffer without harm to security. A VDD that copies the input buffer to an output buffer can ask for a check on the input buffer. Otherwise, the user never sees the contents of the supervisor pages.

- + (Not sure about this one)
Supervisor bits must only be checked on writes when pages are between 1Mb+64K and 4Mb.

PRO:

On a 486 can do writes without checking page permissions. (The 386 is fouled up for read only and must be checked anyway). This would speed up em86. Apps will not be allowed to have their stacks between 1Mb+64K and 4Mb. That will automatically cause a fault. Once

that was checked for, page permission checks could be skipped on a 486 (catching read-only faults).

VDDs with large buffers can check their location and skip supervisor checks if they are entirely above 4Mb.

CON:

If a large amount of secure data is required for a VDD and it can't fit below 4Mb, the VDD would have to put the data in the system arena.

Allows Broken/malicious programs to write on supervisor pages above 4Mb by passing a buffer to a VDD that writes to user space or by setting its stack on top of supervisor pages.

Assumes caller has already checked sel is ring 3 LDT.
When it is known the address is in the V86 address space, this should not be called.

```
BOOL VDHENTRY VDHReadUbuf(pDstBuf, cb, sel, pulOffset, fl)
BOOL VDHENTRY VDHWriteUbuf(pDstBuf, cb, sel, pulOffset, fl)
```

```
fl -- controls on what to check for
R/W      -- Fault if R/W bit set
S/U      -- Fault if S/U bit set
CheckSelWrite -- Fault if descriptor is read-only
SelIsPresent -- Caller knows descriptor is present
SelIsWritable -- Caller knows descriptor is writable
```

*pulOffset is updated with the address of the fault.
For selector faults it is unchanged

```
BOOL VDHENTRY VDHReadULin(pDstBuf, cb, pladdr, pfl)
BOOL VDHENTRY VDHWriteULin(pDstBuf, cb, pladdr, pfl)
```

```
fl -- controls on what to check for
R/W      -- Fault if R/W bit set
S/U      -- Fault if S/U bit set
```

*pladdr is updated with the address of the fault.

Callers can:

- + Skip checking when information is known.
- + Skip checking when access is safe even when a violation (supervisor checking on a read).
- + Ask to check write permissions on reads when the same buffer will be written to later. It is usually much easier to unwind when a fault is hit early.

Callers must be able to handle failures:

- + Back out actions performed so instruction can restart (possibly updating client registers so the instruction restarts after what has already been done).
- + Save data and arrange a callback after the exception handler runs to continue the operation. (Service to be specified by Em86)

VDHArmExceptionCallback (hook) - to be specified

Notes for em86 user space accesses.

Places access happens: (need to make list complete)

- + Arming return hooks (stack)
- + VDHPushStack, VDHPopStack (stack)
- + Instruction decoding (code)
- + IO simulation (segment) rep insw, rep outsw
 - Needs to update user esi, edi if fault part of the way through so it restarts at the

right place.
- Don't know this code, so I don't know how
much VDDs need to cooperate.

Must check all services and note which ones are not
available when app is in protect mode.

Category 5 IOCTL Functions

This page is intentionally left blank.

Printer Control IOCTL Commands

This page is intentionally left blank.

Set Spooler Status IOCTL

Category 5

Function 47h

Purpose

To inform the printer device driver of changes in print spooler status for this device

Parameter Packet Format

```
-----  
|      BYTE  Command Information      |  
-----
```

Data Packet Format

```
-----  
|      BYTE  Spooler Status           |  
-----
```

Where

Command Information

is reserved and must be set to 0.

Spooler Status

The spooler status byte is defined as:

00h

Spooler is not active

01h

Spooler is active

02h - FFh

Reserved. Invalid input.

Returns:

in the kernel request packet status field:

0100h

Command completed successfully.

8103h

Invalid Command

Spool Status invalid.

Remarks:

None.

Get Spooler Status IOCTL

Category 5

Function 67h

Purpose

To inform the requester of the status of the print spooler for this device

Parameter Packet Format

| BYTE Command Information |

Data Packet Format

BYTE	Spooler Status
------	----------------

Where

Command Information

is reserved and must be set to 0.

Spooler Status

On return Spooler Status is defined as:

00h

Spooler is not active

01h

Spooler is active

02h - FFh

Returns:

in the kernel request packet status field:

0100h

Command completed successfully.

Remarks:

None.

MVDM Workbook

Issued by:
John Tyler
ESD OS/2 CP Design
Department 2R2
Building 227-1
Internal Zip 3660
Boca Raton, FL 33429
Tie line: 982-9220
VNET: JTYLER2 at BCRVMPC1

Multiple Virtual DOS Machines Architecture

This page is intentionally left blank.

MVDM Architecture

This page is intentionally left blank.

Problem Description/Objectives

This page is intentionally left blank.

Market Goals

The key goal of MVDM is to make OS/2-386 a "better DOS than DOS" itself. This will make OS/2-386 the operating system of choice on 386-based PCs, and help OS/2 win the battle against Unix, and ensure for itself the undisputed role as the successor to DOS.

Toward this end, OS/2-386 must also match up well against other multi-tasking virtual 8086 (V86) mode environments, notably Windows/386 and DesqView/386.

From the perspective of a user, DOS applications will appear to behave exactly like VIO applications:

1. May run either full-screen or in a window
2. Can only run graphics mode in full-screen
3. Run in background if doing text screen output
4. Windowed DOS applications have all the same Shield controls in the System menu, e.g., Mark, Copy, Paste, Large/Small Font.

Furthermore, DOS applications have some advantages over VIO applications:

1. Can be switched between windowed and full-screen while running
2. A full-screen graphics-mode DOS application can be switched into a window, and the graphics bit-map will be rendered in the window. This allows the Shield layer to support copying graphics to the PM Clipboard, and gives the viewer more context when running multiple applications.
3. For *single-plane* graphics modes (CGA, and VGA 320x200), DOS applications *will* execute in a window and while in the background.

NOTE: We are currently evaluating the support of multi-plane (i.e., EGA and VGA) graphics modes for inclusion in Cruiser.

Key restrictions for DOS applications as compared with VIO applications:

1. Cannot be used in process subtrees

Given these goals, the following key design points were chosen:

1. MVDM
2. Run DOS applications in PM windows, as is done with VIO apps
3. Switching between DOS applications (via PMSHELL)
4. Maximum free memory for DOS applications (600K+)
5. LIM Expanded Memory Manager emulation
6. Basic Clipboard support
 - Copy full-screen graphics
 - Copy full-screen text

OS/2-386 has the following restrictions:

1. No background graphics
2. No DOS block device drivers
3. No direct manipulation of hard disk data

These restrictions arise because the restricted activities are at odds with the multi-tasking OS/2 file system architecture.

4. No DOS network drivers

This restriction exists because the OS/2-386 DOS Emulation does not look like DOS, in terms of its implementation. Hence, a DOS network driver would not be able to install correctly.

PM Clipboard Support

When a VDM is running Full-Screen, the user can switch to the PM screen group, select the System menu on the VDM icon, and select the "Copy All" menu item. This copies the VDM video buffer to the PM Clipboard in a PM format. If the VDM is in a text video mode, then text is placed in the clipboard. If the VDM is in a graphics video mode, then a PM bitmap (possibly color) is placed in the clipboard.

When a VDM is running in a window, the Mark, Copy, Copy All, and Paste menu options are available. For copying, Mark and Copy work as they do for VIO windows in text mode. Since a VDM window may have graphical content, marking mode supports marking pixel-granular rectangles in this case. Copy All is a short-cut that speeds copying the full video buffer image to the clipboard.

Paste works much like VIO window equivalent. If there is text in the PM clipboard (any other format is ignored), it is sent to the Video VDD, which in turn passes it on to the Keyboard VDD, which translates character codes into hardware scan codes (with the help of the Physical Keyboard device driver), and then simulates those scan codes into the VDM

Solutions/Justification

This page is intentionally left blank.

Supported DOS Environment

The following features are provided to DOS applications:

- All documented DOS interfaces
- Some undocumented DOS interfaces
- Most ROM BIOS Services
- LIM Expanded Memory Manager interface
- XMS (Extended Memory Specification)
- Direct manipulation of common hardware devices

The following features of DOS are NOT supported by OS/2-386:

- DOS block device drivers (i.e., disk device drivers)
- Manipulation of internal, undocumented DOS data structures.
- DOS network redirectors

Default VDM configuration parameters may be specified in config.sys. Using the PM Shell, VDM configuration parameters may be specified on a per-DOS application basis.

High-Level Structure

MVDM is made up of the following logical components:

VDM Manager	Creates, initializes, and destroys VDMs.
VDD Manager	Loads, initializes, and communicates with VDDs Provides VDH services to VDDs, analogous to the Device Help routines provided to 16-bit PDDs
8086 Emulation	Manages communication between 8086 instruction streams and VDDs. The most important aspects of this communication is routing software interrupt and IN/OUT instruction traps to the appropriate VDD
DOS Emulation	Supports all documented and some undocument DOS interfaces.
Installable VDDs (VDDs) are used to virtualize hardware devices and the ROM BIOS.	

MVDM Initialization

At OS/2-386 system initialization time, the VDM Manager is initialized. It in turn initializes the 8086 Emulation component and then loads and initializes the VDDs. An initial V86 mode memory map is constructed at this time for later use at VDM creation. Configuration data is obtained from config.sys and OS2.INI.

VDM Creation

When the user starts a DOS application, the VDM Manager notifies the VDDs, giving them an opportunity to do per-VDM initialization. Memory is allocated and mapped into the V86-mode address space, and the DOS Emulation code is initialized. Control is passed to the DOS Emulation kernel to load the DOS shell (usually COMMAND.COM) and the application selected by the user.

VDM Screen Switching

The OS/2 Session Manager notifies the VDM Manager of screen switches. The VDM Manager, in turn, notifies all VDDs which are interested in these events (generally the video, keyboard, and mouse VDDs). The Video VDD (VVIDEO) will take appropriate steps to map in physical/logical video RAM and disable/enable I/O port trapping for the video hardware. The virtual keyboard VDD (VKBD) may set some flags to control the reaction to keyboard polling behavior.

VDM Destruction

When a user exits a VDM, the VDDs are notified so they may clean up any resources owned by that VDM. Memory is released and the process is killed, just as with a normal OS/2 process. VDMs may also be terminated by: 1) a VDD if it detects unsupportable behavior, or 2) by the DOS Emulation component, if it gets corrupted or is otherwise unable to function.

VDM Robustness

MVDM is designed such that only non-trusted code resides in V86 mode. As such, not only is there a great deal of free memory for DOS applications (over 600K compared to a usual 570K for DOS), but OS/2-386 is immune to damage by DOS applications.

IOctls, DOS Applications, and 16-bit OS/2 Physical Device Drivers

In OS/2, INT 21 IOctI (AH=44h) calls are passed to the OS/2 device driver in REAL MODE. In OS/2-386, these IOctI calls are passed to the OS/2 device driver in PROT MODE (since we ALWAYS run these in PROT MODE -- this is a protection feature).

This is all fine and dandy, until you realize that the IOctI packet may CONTAIN pointers. Note that the addresses of the IOctI packets themselves are translated by the kernel, since it knows they are pointers. However, the kernel has no way to determine if there are pointers INSIDE the packets.

One solution (that was quickly discarded) was to imbed the knowledge of all the system-defined packet formats in the kernel, and have the kernel do translation. We dropped this solution because it is inelegant, and, more importantly, does not solve the problem for third-party PDDs.

The obvious solution is to assign a new "device level" for OS/2-386 PDDs, and only support IOctIs from a VDM to these new level PDDs. IOctIs to old PDDs will return an error.

Note that this isn't much of a hardship, since we will convert all of the base PDDs to new level PDDs. Few DOS apps issue IOctIs directly to devices. The Get/Set device attributes IOctIs are handled by the DOS Emulation Kernel, so they do not apply.

The large majority of IOctIs issued in a VDM will come from OS/2 utilities if we continue to support them in a VDM!), and these go to the OS/2 base PDDs.

When a new level PDD gets an IOctI, it must check the info seg to determine if the calling process is a VDM. If it is, then it knows to interpret pointers in packets (if any exist), as real-mode segment:offset addresses. The PDD can compute the linear address directly (segment<<4 + offset), and then use a few of the new DevHlp services to construct a 16:16 address.

VDM Interrupt Support

This page is intentionally left blank.

Hardware Interrupts

In pre-Cruiser releases of OS/2, DOS applications were restricted from hooking hardware interrupts when that IRQ level is owned by an OS/2 device driver. A DOS application that violated this rule was terminated. An Exception to this rule was made for keyboard (the OS/2 Keyboard Device Driver shares the interrupt level with the DOS application).

In Cruiser, this restriction is significantly relaxed. Depending on the support provided by the device's VDD, DOS applications may hook the hardware interrupt vector. The following list summarizes the support provided by Cruiser VDDs for each of the hardware interrupt request (IRQ) levels:

IRQ 0 - Timer (INT 08H)

The INT 08H handler is invoked on HW interrupts from Channel 0 of the system timer. DOS applications may hook this interrupt.

See the description for INT 08H in [Reference](#)

IRQ 1 - Keyboard (INT 09H)

The INT 09H handler is invoked upon the make or break of every keystroke. DOS applications may hook this interrupt.

See the description for INT 09H in [Reference](#)

IRQ 2 - Cascade (Slave) Interrupt Controller

Supports interrupt request levels 8 to 15, described below.

IRQ 3 - Serial Port (COM2, COM3)

Supported when the VCOM and COM0x.SYS are installed.

See the section on VCOM for more detailed information.

IRQ 4 - Serial Port (COM1)

Supported when the VCOM and COM0x.SYS are installed.

See the section on VCOM for more detailed information.

IRQ 5 - Reserved

A VDD to simulate this HW interrupt is not provided in Cruiser

See [Reference](#), for further information.

IRQ 6 - Diskette

In Cruiser, this HW interrupt is not simulated in VDMs.

See the section on VDSK for more detailed information. Also, see the description for INT 0EH in [Reference](#)

IRQ 7 - Parallel Port

In Cruiser, this HW interrupt is not simulated in VDMs.

See the section on VLPT for more detailed information.

IRQ 8 - Real Time Clock

In Cruiser, this HW interrupt is not simulated in VDMs.

See the section on VCOM for more detailed information. Also, see the description for INT 70H in [Reference](#)

IRQ 9 - Redirect Cascade

A VDD to simulate this HW interrupt is not provided in Cruiser

See [Reference](#), for further information.

IRQ 10 - Reserved

A VDD to simulate this HW interrupt is not provided in Cruiser

See [Reference](#), for further information.

IRQ 11 - Reserved

A VDD to simulate this HW interrupt is not provided in Cruiser

See [Reference](#), for further information.

IRQ 12 - Auxiliary Device

A VDD to simulate this HW interrupt is not provided in Cruiser

See [Reference](#), for further information.

IRQ 13 - Math Coprocessor (NPX)

NPX exception interrupts on IRQ 13 are reflected into the VDM.

See the section on VNPX for more detailed information.

IRQ 14 - Fixed Disk

In Cruiser, this HW interrupt is not simulated in VDMs.

See the section on VDSK for more detailed information.

IRQ 15 - Reserved

A VDD to simulate this HW interrupt is not provided in Cruiser

See details in [Reference](#), below.

Reserved IRQ support in Cruiser

Cruiser does not provide support for optional devices on the "Reserved" and "Redirect Cascade" interrupt levels, either in OS/2 mode sessions or in VDMs.

Hooking one of these HW interrupt request levels in a DOS application has no effect. Although the hardware interrupts may actually be generated, if there is no PDD-VDD pair to service that interrupt, the DOS application's interrupt handler will not be invoked.

These hardware interrupt request levels may be supported in a VDM by a user-installed VDD. A user-installed PDD is also required to provide this support. There is no mechanism for a VDD to directly service a hardware interrupt. The PDD must act as an intermediary.

VMCB Software Interrupts

The following list documents the VMCB software interrupt compatibility considerations and restrictions for Cruiser VDMs. This list is provided as a companion to, not as a replacement for, the IBM PS/2 and Personal Computer BIOS Interface Technical Reference (Publication # 68X2341).

Note: *This document lists only specific compatibility considerations and functional restrictions for users of VMCB in the VDM environment of Cruiser. For VMCB services not listed below, no known restrictions apply.*

02H - Non-maskable Interrupt (NMI)

The NMI in a VDM has little meaning. Currently, the only source of the NMI is a memory fault. Such indications should continue to be reported to the OS/2 kernel. Non-maskable interrupts are not reflected to the VDMs. This VMCB interrupt handler is therefore not invoked on NMI.

See the section on VCOM for more detailed information.

05H - Print Screen

The VMCB INT 05h handler is supported.

If the DOS application issues INT 05h, the contents of the video buffer will be printed. The Shift-PrtSc keystroke combination (or equivalent) will also invoke this function.

Note: The Spooler must be active in order to enable printing from a VDM.

See the section on VLPT for more detailed information.

08H - System Timer

This interrupt is invoked every time the system timer channel 0 counts down to zero (normally, 18.2 times per second). DOS applications may hook this interrupt. In a VDM, however, these Timer interrupts may come in a burst and have variable latency. The INT 08H interrupt handler is emulated by the VTDD.

See the section on VTIMER for more detailed information.

0EH - Diskette

The INT 0EH handler is not used to service interrupts generated by the diskette device (IRQ 6). DOS applications should not hook this interrupt. See the section on VDSK for more detailed information.

10H - Video

The INT 10H functions are fully supported in VDMs. The following INT 10H functions are emulated by the VVD

- 0EH - Write Teletype (TTY)
- 13H - Write String

Note: These functions are emulated only if the INT 10h vector is unmodified. If any piece of VDM code hooks INT 10h, then emulation occurs only if these functions are passed on to the VVD's interrupt hook. See the section on virtual video drivers for more detailed information.

13H - Disk/Diskette

INT 13H emulation in VDMs supports both removable and non-removable media.

The following subset of INT 13H functions are supported in VDMs:

- 00H - Reset Diskette
- 01H - Read Status
- 02H - Read Sectors
- 03H - Write Sectors (Diskette only)
- 04H - Verify Sectors
- 05H - Format Track (Diskette only)
- 08H - Get Drive Parameters
- 0AH - Read Long (Fixed Disk only)
- 15H - Read DASD Type
- 16H - Change Status (Diskette only)
- 17H - Set Disk Type (Diskette only)
- 18H - Set Media Type (Diskette only)

See the section on VDSK for more detailed information.

14H - Async

When the VCOM is installed, all VMCB INT 14H functions are supported. The INT 14H functions are emulated by the VCOM to enhance performance.

See the section on VCOM for more detailed information.

15H - System Services

AH	Function	Comments
00	Cassette Motor On	DEFAULT - ROM BIOS will handle it.
01	Cassette Motor Off	DEFAULT - ROM BIOS will handle it.
02	Cassette Read	DEFAULT - ROM BIOS will handle it.
03	Cassette Write	DEFAULT - ROM BIOS will handle it.
0F	Format Periodic Int	ERROR - VBIOS manages this. On return, CF is set. NOTE: Do not allow ESDI format command.
4F	Keyboard Intercept	DEFAULT - ROM BIOS will handle it.
80	Open Device	DEFAULT - ROM BIOS will handle it.
81	Close Device	DEFAULT - ROM BIOS will handle it.
82	Program Terminate	DEFAULT - ROM BIOS will handle it.
83	Event Wait	SUPPORT - VTIMER supports this. AL = 0, set timer CX:DX = ULONG microsecond timeout ES:BX = ptr to byte, high order bit is set when timeout expires. AL = 1, reset timer NOTE: ints are enabled during this wait.
84	Joystick Support	DEFAULT - ROM BIOS will handle it.
85	SysReq Key Pressed	SUPPORT - VKBD issue this. On return, AL = 0, key make AL = 1, key break
86	Wait	SUPPORT - VTIMER supports this. CX:DX = ULONG microsecond timeout On return, CF set if wait in progress CF clear otherwise NOTE: ints are enabled during this wait.
87	Move Block	ERROR - VXMS manages this. On return, AH = 01, RAM Parity Error. NOTE: No real DOS app would call this function after finding out there is no Extended Memory (by calling AH=88).
88	Get Extended Mem Size	ERROR - VBIOS/VXMS manages this. On return, AX = 0, no extended memory
89	Switth to Prot Mode	ERROR - VBIOS manages this. On return, AX != 0, fail switch.
90	Device Wait	DEFAULT - ROM BIOS will handle it.
91	Device Post	DEFAULT - ROM BIOS will handle it.
C0	Get System Config Parmes	DEFAULT - ROM BIOS will handle it.
C1	Get EBIOS Data Area	DEFAULT - ROM BIOS will handle it.
C2	PS/2 Mouse Functions	SUPPORT - VMSE will manage this. On return, CF is clear (no error) CF is set (error condition) AH = error code.
C3	Watchdog Timeout	IGNORE - VBIOS will manage this. On return, CF is clear. NOTE: No harm in ignoring this.

```
C4 Prog Option Select      ERROR - VBIOS will manage this.
                           On return,
                           CF is set.
```

See the section on VBIOS for more detailed information.

16H - Keyboard

Supported directly by the ROM VMCB INT 16H routines.

See the section on VKBD for more detailed information.

17H - Printer

Supported (emulated) by the VLPT

See the section on VLPT for more detailed information.

19H - Reboot

Supported - However, this does not operate in the same manner as DOS. The system is not restarted. Rather, the VDM is terminated and a new VDM session is started.

1AH - Time Of Day

The VCMOS component supports read-only access to the Real Time Clock device. Because of this restriction, the following VMCB INT 1AH functions are not supported in VDMs:

- 01H - Set RTC Count
- 03H - Set RTC Time
- 05H - Set RTC Date
- 06H - Set RTC Alarm
- 07H - Reset RTC Alarm
- 08H - Set RTC Activated Power-On mode
- 0BH - Set System-Timer Day Counter
- 80H - Set Up Sound Multiplexer

A DOS application may read the RTC Count, Time, and/or Date. The INT 1AH functions are not emulated by the VCMOS. Rather, the VMCB ROM functions are executed directly. The RT/CMOS hardware is virtualized by the VCMOS.

See the section on VCOM for more detailed information.

1EH - Diskette Parameters

The segment address of the diskette parameters table is stored in the IVT 1EH entry.

See the section on VDSK for more detailed information.

70H - Real-Time Clock Interrupt

Interrupts from the RTC, IRQ 8, are not reflected into VDMs. This software interrupt handler is not used for either alarm or periodic interrupts.

See the section on VCOM for more detailed information.

DOS Software Interrupt Support

The DOS emulation component supports the documented aspects of the following DOS features:

See the section on DOSKRNL for more detailed information.

- 20H - Program terminate
- 21H - DOS Function Request

All documented INT 21H functions, plus some undocumented INT 21H functions, are supported by DOS emulation. However, the following functions are supported with restrictions:

- 38H - Return Country Dependent Information
- 44H - Generic IOCTL
- 66H - Get/Set Code Page

- 67H - Set Handle Count

See the section on DOSKRNL for details on the restrictions.

- 22H - Terminate Address
- 23H - Ctl-Break Exit Address
- 24H - Critical Error Handler
- 25H - Absolute Disk Read
- 26H - Absolute Disk Write

A hard error is reported on requests for non-removable media.

- 27H - Terminate but Stay Resident
- 28H - Idle Loop
- 2Fh - Multiplex

Time slice yield (AX = 1680h). When a DOS application issues int 2fh with AX = 1680h, it offers to yield its time slice. This can be used by DOS applications in busy wait loops. (Preserves all registers).

Video services. See the section on virtual video drivers for more detailed information.

Print Spool Interface

Other Software Interrupt Support

The following are other software interrupts supported in the VDM environment of Cruiser.

INT 33H - Mouse

Supported - When the Cruiser VMD is installed, INT 33H functions are available.

See the section on virtual mouse device handler for more detailed information. Also, refer to the *Microsoft Mouse Programmer's Reference Guide* for a complete functional description of the INT 33H mouse interface.

INT 67H - LIM Expanded Memory Manager (EMM)

Supported - When the Cruiser Expanded Memory Manager VDD is installed, the Lotus/Intel/Microsoft (LIM) EMM v4.0 INT 67H functions are available.

See the VEMM description for more detailed information. Also, refer to the [Lotus/Intel/Microsoft Expanded Memory Specification Version 4.0](#) for a complete functional description of the INT 67H EMM interface.

VDM Manager Architecture

This page is intentionally left blank.

Problem Description/Objectives

The VDM Manager contains the mechanism to start and interact with DOS applications. The environment in which a DOS application executes is called a VDM (VDM). VDMs run in the Virtual 8086 (V86) mode of the Intel 80386 CPU. VDMs are created, initialized and destroyed by this component. The VDM Manager is responsible for loading and initializing all virtual device drivers. Virtual Device Drivers are required to virtualize the hardware and ROM BIOS.

- ROM BIOS services
- Direct manipulation of Video RAM
- Direct manipulation of the ROM BIOS data area
- Direct programming of I/O ports
- Intercepting hardware interrupts
- Intercepting software interrupts

The VDDs must support all of the above behavior for multiple DOS applications running simultaneously in the system.

In general, our goal is to support as many different DOS applications as possible.

- Provide compatible support for all standard hardware
- Support installable virtualization, so that new hardware may be added in the field without requiring an OS upgrade.

The general architecture of VDDs and different phases of their loading and initialization are described in this document.

These VDD's use virtual device help services at initialization time, VDM creation time and task time. These services are provided by different components including the VDM Manager. Following is a brief list of VDH services provided by different components. Please refer to the respective component for a service description.

- Memory and Address Space Management Services
- PDD-VDD-OS/2 Communication Services
- PageFault Services
- VDM State Control Services
- Event Management Services
- Semaphore Services
- 8086 Emulation Services

The VDM Manager component provides the following services:

VDM Manager Services:

- VDM creation.
- VDM destruction.
- VDM configuration information.
- Per-VDM VDD instance data management.
- Page fault routing

VDH Services:

- Memory management services:
 - Allocation/Reallocation/Freeing services for:
 - both global and per-VDM objects
 - Page or Byte granular objects
 - options for fixed, swappable allocations
 - Allocation of memory in DOS arena.
 - Reserve specified linear space.
 - Mapping services:
 - Map to physical address.
 - Map to linear address.

- Map to invalid pages.
 - Map to black holes (don't care) pages.
- Copy/Exchange services.
- Block management services (pools of equal-size memory blocks)
- Query services:
 - Query the biggest linear space in a specified range.
 - Query dirty bits for set of pages.
 - Query amount of free virtual memory.
- OS/2-process-to-VDD communication
- VDD/PDD communication
- VDM event list management
- Page fault handler registration
- Error message and Display
- Terminate VDM service.
- Mutual Exclusion and Event semaphores
- Freeze/Thaw services
- Timer/Priority/Query
- V86-mode stack manipulation
- Idle DOS application management

MVDM Initialization

At init time, the VDM initialization routine is called to initialize the VDM manager. SysInit passes a list of VDDs to be loaded. "base" VDDs are supplied to SysInit from OS2LDR. These are VDDs which must load successfully for MVDM to be functionally. Installable VDDs are supplied in "device=" lines from CONFIG.SYS. MVDM will still be operational if these are not loaded (except for the video VDD, which must load).

All of them are loaded and initialized, one by one, in the order specified in the list. VDDs do global initialization only at this point, including reserving device memory.

With the help of these reservations and other boot time configuration information, a global memory map is created. This is an initial configuration used when creating a VDM.

VDM Creation

The Shell (via the Session Manager) calls DosCreateVDM to create a VDM. This API may only be called by the Shell. This API accepts the name of a DOS program to run, along with several parameters to customize the VDM:

- Amount of VDM memory (64K..640K).
- Number of FCB SFTs immune to recycle, and total number of FCB SFTs.
- BREAK flag
- VDD Properties
- List of DOS device drivers to install

DosCreateVDM creates a process, marks it as a VDM, and creates a 0-4Mb linear address space. Next, 8086 emulation is initialized. Next, all registered (via VDHInstallUserHook) VDM Creation hooks are called, to allow VDDs to do per-VDM initialization. Finally, the DOS Emulation kernel (DOSKRNL) is loaded, and we return to V86 mode to let the DOS Emulation kernel initialize and load the user application.

The VDM process does not inherit file handles from the parent process. The VDM process does inherit current drive and directory, but those apply only after AUTOEXEC.BAT is executed.

VDM Termination

VDM termination happens when the application finishes, or when a VDD terminates the VDM due to some illegal operation. All registered VDM termination hooks are called, then 8086 emulation cleans up its per-VDM resources, then the VDM Manager cleans up its per-VDM resources, and finally the VDM is destroyed (as with a normal OS/2 process).

Page Faults in a VDM

Any time a page fault happens on an invalid page in VDM memory, the fault is routed to the handler that hooked the particular linear address (via VDHInstallFaultHook). If no handler has registered for the faulting address, the page is mapped to don't care pages. This is compatible with real hardware.

This page fault service is provided to VDDs to do allow "lazy" or "copy on demand" memory management. The video VDD is the primary client among the standard VDDs.

Virtual Device Drivers

VDDs (VDDs) are installable modules responsible for virtualizing the hardware and ROM BIOS aspects of the DOS environment for VDMs. They are responsible for the following features:

- Maintain virtual hardware state for each VDM (VDM)
- Prevent a VDM from corrupting the state of another VDM, or the system as a whole
- Support fast screen I/O
- Support fast communications I/O

The following is the general architecture of VDDs. PLEASE see the specific VDD design section for details on the design of a particular VDD.

Virtual Device Driver Architecture

This page is intentionally left blank.

Problem Description/Objectives

A Virtual Device Driver (VDD) is responsible for virtualizing a particular piece of hardware and associated ROM BIOS in the manner expected by a DOS application. It must do so in a compatible fashion, with minimal performance degradation, and without compromising the integrity of the system. To achieve a certain level of hardware independence, a VDD may communicate with a Physical Device Driver (PDD, a.k.a. an OS/2 Device Driver) to interact with hardware. At a minimum, this can serve to mask the differences between an ISA, MCA, and EISA hardware.

Design Goals

The responsibility of a VDD is to manage I/O ports, device memory, and ROM BIOS services for the device(s) which it virtualizes.

Hardware compatibility

The key rule here is that the VDD should behave, as much as is possible, exactly like the physical hardware. Straying from this rule is likely to result in incompatible emulation.

Hardware independence

Where possible, the VDD should use a PDD to manipulate hardware. This is the case for VKBD, VMOUSE, VLPT, VCOM, VDSK, and VTIMER. VVIDEO, on the other hand, talks directly to the video hardware, as this is the only way to achieve excellent performance.

ROM BIOS compatibility

Where possible, the VDD should let the ROM BIOS code on the system do its work. The VDD achieves virtualization by emulating I/O port and device memory operations.

The exception to this guideline is where the performance degradation due to I/O and interrupt trapping is unacceptable. For example, the Video VDD intercepts the ROM BIOS video interrupt (INT 10h, usually) and performs the requested operation directly.

DOS Compatibility

In general, if a VDD does a good job of virtualizing the hardware and ROM BIOS, DOS compatibility will just fall out. There may be cases where some understanding of DOS kernel or application behavior can make the VDD exhibit better performance characteristics.

For example, there is a fast-path call from the DOS CON (console) device driver to the video VDD which speeds up writing to the screen.

Because of this, Cruiser is 3-6 times faster than DOS when TYPEing a file in COMMAND.COM. To maintain compatibility, however, this method is used only when the ROM BIOS INT 10h (video) vector has not been hooked by code in the VDM.

Performance and Size

Memory is a precious resource, so VDDs should attempt to minimize the amount they consume. The dword-granular VDHAllocMem makes it convenient to allocate only as much memory as is required for a particular use, so VDDs need not reserve arrays of per-VDM structures (as PDDs did in 286 OS/2 because they had no choice). VDHAllocPages can be used to allocate swappable memory, so a VDD need not increase the amount of resident space consumed. VDDs should use initialization code and data objects for work that is only done at VDD installation. Resident memory should only be used for code and data that must be accessible at hardware interrupt time (i.e., when called by a Physical Device Driver

Performance is an especially important concern because of the inescapable overhead of trapping from V86 mode to Ring 0 on ALL software interrupts and on hooked I/O ports.

A VDD designer should pinpoint those areas of a VDD that must be fast, and then orient the design around those areas. Performance-critical code should be written in assembler. Any other code should be written in C to maximize maintainability and portability.

Protection

A VDD is inherently protected from a VDM because it is not visible in the VDM address space. However, a VDD must be careful to check all parameters coming in from a VDM to ensure that it does not damage itself or some other part of the system.

When a VDD detects an invalid operation, it should kill the offending VDM. The section below on VDD Errors provides specific recommendations.

A VDD may also receive parameters from an OS/2 application via DosRequestVDD. The VDD must use VDHLockMem to verify the addressability of OS/2 application buffers.

The kernel preserves only EBX, ESI, EDI, EBP, DS and ES across VDH service calls. Similarly, VDDs must preserve those registers whenever they are called by the kernel, regardless of whether the call is to a C hook or an assembly hook. In general, EAX, ECX, EDX, FS, GS and FLAGS should be considered volatile across any procedure call, with the exception of the Direction bit in the FLAGS register. On entry to and return from any procedure, the Direction bit must be clear, and CS, DS and ES must contain the Ring 0 selectors that map the FLAT code/data address space. Page faults and other non-fatal exceptions and interrupts always preserve all registers.

VDDs are not allowed to access any of the System Registers, including GDTR, LDTR, IDTR, TR, CR0-3, DR0-7 and many of the bits in the FLAGS register. The only FLAGS bits a VDD may access are the Arithmetic bits, the Interrupt bit, and the Direction bit. Care must be taken when clearing the Interrupt bit; it is a fatal error to attempt to execute code or access data that is not resident when interrupts are disabled.

With regard to a VDM's registers as they exist on a VDM stack frame, they may be freely examined or modified, with the exception of the VDM's FLAGS, which should be modified with VDHSetFlags only. Direct modification, if required, must be restricted to the Arithmetic bits and the Direction bit.

Standard VDDs

These VDDs are supplied with Cruiser:

VDD	Description
vbios	BIOS - miscellaneous ROM BIOS support
vcmos	CMOS - manages CMOS data area
*vcom	Async
vdma	DMA
*vdsk	Disk - only for INT 13 copy-protection
vemm	EMM - LIM 4.0 Expanded Memory Manager

*vkbd	Keyboard
*vlpt	Printer
*vmse	Mouse
vnpx	NPX – Numeric Processor eXtension (80387)
vpic	PIC – Programmable Interrupt Controller
*vtimer	Timer
vvideo	Video – Depends on hardwar (VCGA, VEGA, VVGA, V8514, etc.)
vxms	eXtended Memory Specification

* = VDD which talks to corresponding PDD.

VDD Structure

A VDD is a 32-bit EXE file. It can have zero or more objects of the following types:

1. Initialization code
2. Initialization data
3. Swappable Global code

There must be at least one object of this type.

4. Swappable Global data
5. Swappable Instance data
6. Resident Global code
7. Resident Global data
8. Resident Instance data

The resident objects should be used only for code and data that must be accessible at physical hardware interrupt time (when a PDD calls the VDD). A VDD which does not interact with a PDD needs no resident objects (VVIDEO, VEMM, and VXMS are such VDDs).

VDD EXE Object Limitations

In order to support interaction with 16-bit PDDs, VDD data objects are limited to 64Kb in size. This allows the VDM manager to conveniently allocate 16-bit selectors to map the data, and makes the translation from 0:32 pointers to 16:16 pointers very efficient.

Multiple objects of each type are allowed, and dynamically allocated objects can be larger than 64K, so this restriction should not be too severe.

Environment

Though VDDs execute at Ring 0 along with the kernel, PDDs, FSDs, and other Ring 0 components, they are logically isolated to a specific view of the system.

Memory

The VDM private address space starts at linear address 0, and may grow upwards until it runs into the system address space. The first 1Mb+64Kb of the VDM address space is the "v86 address space". This is the area where the DOS Emulation Kernel and DOS Applications reside, as dictated by the v86 mode of the 80386.

The "system area" is global address space that starts at 4Gb and grows downward. System area memory objects are addressable in *any* process context (be it a VDM or an OS/2 process), whereas private area memory is addressable only in the context of the particular process (but see the discussion of HVDM below for an exception).

HVDM - VDM Handle

Each VDM is uniquely identified by a VDM handle. This is a linear address in the system area which aliases the first 4Mb of the VDM.

The HVDM is used on many of the VDH calls to identify a particular VDM. It may also be used to address per-VDM data when the target VDM is not current process. This is especially convenient for accessing VDD instance data, which the VDM Manager locates below the 4Mb boundary (but above the v86 address space!) to allow for out-of-context access. For example, this allows the keyboard VDD to have access to a per-VDM buffer at physical hardware interrupt time regardless of which process is currently running (assuming this per-VDM buffer is in a resident object!)

Stack

The VDD executes on one of two stacks, depending upon whether it is executing at "task-time" or "interrupt-time".

Task-time is the common case, where the VDD has received control due to a trap from v86 mode. In this case, the VDD is running on the "kernel stack" of the VDM which caused the trap. Due to the large amount of 16-bit code in the kernel, ESP must be below 64K so that SS:SP and SS:ESP always point to the same physical memory.

The SStoDS macro *must* be used when passing the address of a frame (automatic) variable in a function call, since 32-bit code assume SS==DS.

Interrupt-time occurs when the system is processing a physical hardware interrupt. The only way for a VDD to get control at interrupt time is to be called by a PDD with which it had set up communication. At this time, the VDD will be executing on the "interrupt stack" which is maintained by the kernel. This stack is also limited to ESP being less than 64K, since 16-bit PDDs must be able to work with SS:SP.

Segment registers

VDDs are compiled such that SS is assumed to always be equal to DS. The Ring 0 DS and SS are not equal, though (see above), so VDDs must behave peculiarly in certain situations.

In C code, the only time this difference matters is when passing pointers to automatic variables to another function. The SStoDS() function/macro must be called to convert the SS-relative 32-bit offset to a DS-relative 32-bit offset. This applies to LocalVars in MASM code as well.

In MASM, the programmer must be explicitly aware that SS != DS and avoid any incorrect code. In most cases, if MASM code is written in a fashion similar to C (with LocalVars), the SStoDS() macro should be the only special technique required.

The VDD must also understand segment registers when passing parameters to 16-bit PDDs. Please see the section below on VDD/PDD interaction for details.

VDH Services

The VDM Manager supplies a set of Virtual Device Help (VDH) services to provide an abstract but efficient means for VDDs to interact with VDMs and with the OS/2 kernel.

VDH Services have the following characteristics:

1. Are available via dynamic linking.
2. Use the 32-bit PLM calling convention.
3. A return value of 0 (FALSE) usually means the call failed. When FALSE is returned, calling VDHGetError will return a detailed error code. If VDHGetError returns FALSE, then the last call succeeded, and 0 was a meaningful return value (not an error).

A return value of non-zero (TRUE) means the call succeeded.

These return value semantics allow us to reduce the parameter count by one in many cases, thus improving the speed of the call.

4. All pointer parameters are 0:32 flat pointers.

The following services are used to manage page granular blocks of linear space on behalf of a VDM:

- VDHFindFreePages - find a region of free linear space below 1M+64K
- VDHReservePages - reserve region of linear space below 1M+64K
- VDHUnreservePages - unreserve region of linear space below 1M+64K
- VDHQueryFreePages - determine amount of free virtual memory
- VDHMapPages - map allocated pages into reserved region
- VDHInstallFaultHook - install hook for page faults
- VDHGetDirtyPageInfo - read (and clear) dirty page bits
- VDHAllocPages - allocate linear space and commit backing store
- VDHReallocPages - grow/shrink previous page allocation
- VDHFreePages - free previous allocation

These are intended to be used together in the following ways, which match the needs of VDDs:

- Reservation and Mapping
- Reservation and Allocation/Reallocation/Free
- Allocation/Reallocation/Free

Reservation and Mapping



Status of each page (invalid, black hole, mapped to physical memory, mapped to linear memory) is independant of every other page.

In this scheme, the VDD calls VDHReservePages to reserve a range of linear addresses. These are usually associated with a memory-mapped hardware device (for example, the Video VDD reserves the video RAM range), although the LIM emulation VDD uses this

same technique to simulate hardware paging.

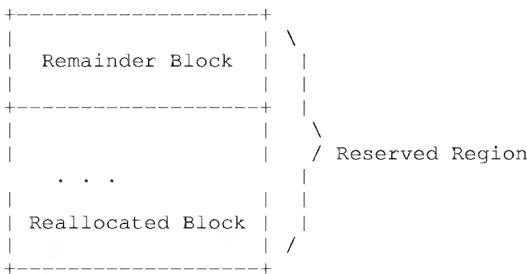
Next, the VDD will use `VDHAllocPages` to allocate memory which is can then be mapped into the reserved region using `VDHMapPages`.

EXAMPLE: When a VDM is in the background, the Video VDD will have allocated a virtual screen buffer, and maps the reserved range to that virtual buffer. When the VDM is brought to the foreground, the video VDD will map the reserved range to the physical video RAM, allowing the application to update the video display directly.

`VDHInstallFaultHook` is used to trap a VDM that touches an invalid page (mades so by `VDHMapPages`). For example, the video VDD uses this to detect when a DOS application has touched video RAM while in a graphics mode, at which point the DOS application is frozen.

`VDHGetDirtyPageInfo` is used to get the dirty page bits (which indicate whether a page has been written to). For example, The video VDD uses this to determine what video RAM has been modified so that it need save only those pages of video RAM that have been changed.

Reservation and Allocation/Reallocation/Free



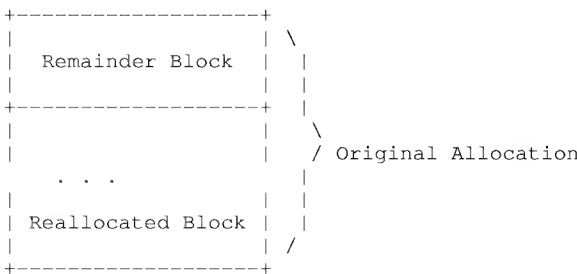
A linear region in the range 0..4Mb is reserved, and memory is allocated/reallocated from the bottom.

In this scheme, the VDD calls `VDHReservePages` to reserve a range of linear addresses. Later, the VDD uses `VDHAllocPages`, `VDHReallocPages`, and `VDHFreePages` to control the amount of memory actually allocated. Memory can only be allocated from the bottom of the region.

This scheme is useful when a VDD wants to allocate memory that can be accessed via the HVDM in any task context, but wants to change the amount of memory allocated, AND does not want to start with the largest allocation initially.

EXAMPLE: The Video VDD reserves a region below 4Mb large enough to hold all of video RAM (256K on a VGA), and then calls `VDHAllocPages` to allocate the smallest amount needed. When a VDM is brought to the foreground, the Video VDD calls `VDHReallocPages` to grow the allocation to the full size of the reservation. When a VDM is sent to the background, the Video VDD saves the physical video RAM to the buffer. If the video card was in a text mode, most of the buffer will be unused, so `VDHReallocPages` is called to shrink the buffer down (to 16K, usually).

Allocation/Reallocation/Free



An allocation can be shrunk and grown, within the size of the original allocation.

In this scheme, an allocation is made anywhere in linear memory. At some later time, the allocation can be shrunk with `VDHReallocPages`. At some even later time, `VDHReallocPages` can be used to grow the allocation. As long as the requested growth does not cause the allocation to exceed its original size, the growth will succeed (assuming there is sufficient swap space available) without changing the allocation start address.

This scheme is generally used outside of the VDM address space, though VBIOS calls `VDHAllocPages` to allocate the DOS arena memory. In this case, `VDHReallocPages` is never called.

Ring 0 details

This section discusses the special aspects of VDDs that are a result of running at Ring 0.

VDD code is always executed at Ring 0. Since code executing at Ring 0 has full access to all CPU instructions, the VDD has a special responsibility to be careful to manipulate only resources that belong to it.

Kernel mode and Scheduler preemption

Prior to giving a VDD control, the VDM Manager always "enters kernel mode". This operation involves setting the "InDos" flag and a few other state variables used by the Interrupt Manager and the Scheduler. The key feature of "being in kernel mode" is that the thread of execution is "not preemptible".

For example, if a VDD executed the following code

```
foo:    jump    foo    ; loop forever
```

the system would effectively stop. That is because the system will not schedule another thread while the InDos flag is set.

Preemption latency concerns

The OS/2 Performance Requirements (see ???) require that, from the point when a new, highest-priority thread becomes runnable, until the point the thread receives the CPU, no more than a fixed amount of time may elapse (see ???). The purpose of this restriction is that it ensures that the system remains responsive to user interaction.

A more reasonable example of actual VDD behavior that could cause excessive dispatch latency is copying large chunks of memory. The Video VDD does this when it does video memory save/restore operations during screen switching. The EMM VDD does this in response to certain EMM function calls.

In order to avoid excessive dispatch latency, a VDD can use the `VDHYield` services. This allows other threads to run. The `VDHCopyMem` and `VDHExchangeMem` services can be used if a VDD wishes to let the kernel worry about this. These are recommended except in cases where some special operations are required (as is the case when the Video VDD is saving/restoring video memory, where the storage formats are different between normal RAM and video RAM).

Interrupt latency concerns

The OS/2 Performance Requirements (see ???) require that interrupts be disabled or postponed for no more than a fixed amount of time (see ???). The purpose of this restriction is that it ensures that the system does not lose hardware interrupt events.

VDDs do not receive hardware interrupts directly; they may only execute at interrupt time when called by a PDD which is at interrupt time. VDD code executed at interrupt time must work speedily and return to the calling PDD.

Many of the VDH services are not available at interrupt time, since they involve operations that may block. Blocking (i.e., waiting for an event to occur or a semaphore to be released) is not possible at interrupt time because the system has no way to switch to another thread.

In general, a VDD should postpone any time-consuming work to task time using VDHArmContextHook.

Initialization

This page is intentionally left blank.

VDD Initialization in the context of System Initialization

VDDs are loaded and initialized after all PDDs are loaded and initialized, but before the Shell is started.

VDD loading

The VDM Manager calls the loader to load VDDs. "Base" VDDs are loaded first, followed by "Installable" VDDs. "Base" VDDs are those identified in OS2LDR that must be present for MVDM to function. "Installable" VDDs are those specified in "device=" lines in CONFIG.SYS. MVDM is still operable if any or all of these fail to load.

The VDD entry point is called after the VDD is loaded. The VDD returns TRUE to indicate a successful load, and it returns FALSE to return an unsuccessful load.

Installing user hooks

Each VDD must use VDHInstallUserHook to install a VDM_CREATE hook. VDDs that care about foreground/background (video, kbd, and mouse) would also install VDM_FOREGROUND and VDM_BACKGROUND hooks.

Allocating global resources

Any global resources should be allocated at this time.

Handling VDD Failures

This page is intentionally left blank.

VDM Termination

When the VDM_TERMINATE hook is called, the VDD is responsible for freeing all resources allocated by the VDD on behalf of the terminating VDM. Please refer to the section titled VDM TERMINATION IMPLICATIONS in the individual VDH service function headers to determine if the VDD is responsible for cleanup, or if it is done automatically by the VDM Manager.

VDDs and Errors

VDDs can experience the following kinds of errors, and should react in the prescribed manner. The examples below are intended to convey the spirit of how a VDD might experience these errors, but these examples do not necessarily describe the actual behavior of the VDDs.

1) Error returned from a VDH service

EXAMPLE:

A call to VDHAllocMem returns 0, meaning that the memory allocation request cannot be satisfied

RESULT:

The VDH service returns a failure indication (FALSE).

REQUIRED VDD BEHAVIOR:

During VDD initialization:

The VDD must fail initialization. It does this by cleaning up any resources already allocated, and then returning FALSE.

During VDM creation:

The VDD must fail the VDM creation. It does this by cleaning up any resources already allocated, and then returning FALSE.

After VDM creation, during a "failable" operation:

The VDD must fail the operation. A "failable" operation is one, initiated by V86 code, that can be failed in a manner such that the v86 understands the operation failed.

EXAMPLE: The first OUT to a printer port may require the allocation of a print buffer. If that allocation fails, the VDD can set some state flags such that the DOS application can determine that the printer is not available.

After VDM creation, during an "unfailable" operation:

The VDD must ask the user what to do.

An "unfailable" operation is one, initiated by V86 code, that cannot be failed in a manner such that the v86 understands the operation failed.

EXAMPLE: This should be a very rare event. No example is readily apparent.

The VDD should call VDHPopup to tell the user what happened, and ask what to do. The allowed responses should be RETRY (after the user had tried to fix the problem) and ABORT (when the user cannot fix the problem). In the ABORT case, the VDD should call VDHKillVDM and return, causing the VDM to be killed.

2) Bad parameter passed to a VDH service

EXAMPLE:

Calling VDHFreeMem on an address which was not allocated by DosAllocMem.

RESULT:

The VDM Manager halts the system.

This error can occur only as a result of a coding defect in a VDD, so returning an error to the VDD is useless. Furthermore, since the VDD is bad, it is impossible to localize the error in any one VDM, or be certain it has not effected kernel system structures.

3) Internal VDD consistency failure

EXAMPLE:

A VDD with a large amount of internal state may find (due to a coding defect) that its state is inconsistent.

RESULT:

The VDD informs the user of the error, and then either terminates the VDM or halts the system.

REQUIRED VDD BEHAVIOR:

If the VDD can localize the error to a single VDM, that VDM should be terminated with VDHKillVDM.

If the VDD cannot localize the error to a single VDM, then the system should be shutdown with VDH HaltSystem. PLEASE NOTE: This is VERY unfriendly behavior. A released, retail VDD should never call VDH HaltSystem.

4) Illegal operation by a DOS application

EXAMPLE:

A DOS application does an OUT instruction to a port controlled by the Disk VDD. The Disk VDD does not support hardware control of the disk controller.

RESULT:

The VDD informs the user of the error, and then either retries the operation, ignores the operation, or terminates the VDM.

REQUIRED VDD BEHAVIOR:

The VDD must inform the user of the problem via VDHPopup and get a response back. The VDD will either retry or ignore the operation, or terminate the VDM with VDHKillVDM.

VDM Creation

Every VDD will want to call VDHInstallUserHook to get notified of VDM creation and termination. These notifications allow the VDD to initialize the virtual hardware state at VDM creation, and to clean up any per-VDM resources at VDM termination.

Allocate per-VDM dynamic data

This is rarely necessary, since VDD instance data is usually sufficient for per-VDM data needs.

A better solution is usually to avoid allocating dynamic per-VDM data until it is necessary. For example, the EMM VDD uses this approach to manage EMM space, which can grow to megabytes in size. Instead of allocating the maximum amount at VDM creation (which would consume horrendous amounts of swap space and slow down VDM creation amazingly), it allocates memory as needed.

Hook I/O ports

This should be done immediately at VDM creation.

Reserve device memory linear address space

This should be done immediately at VDM creation, to ensure that the device can appear in the VDM. The video VDD does this, for example.

Hook page faults

This should be done immediately at VDM creation, if the device being virtualized has device memory on the address bus (e.g., a video controller). Note that the range where page faults are hooked should have been reserved with `VDHReservePages`.

Hook software interrupts

This is generally necessary for VDDs that intercept ROM BIOS interrupts.

Install DOS device driver stubs

This is necessary for VDDs that must supply a "fake" DOS device driver. This can only be done at VDM creation time. After that point, the DOS Emulation component has control over the DOS heap.

VDM Termination

At VDM termination, all termination hooks registered with `VDHInstallUserHook` are called. The VDD at this time should free any global resources associated with the terminating VDM, and should break any connections with PDDs or the shield layer, as appropriate.

VDM Session Switching

A VDD can register hooks to be called when a VDM is switched between foreground and background. See the section on the video VDD for details about session switching.

Interacting with another VDD

Since dynamic linking is supported between VDDs, a solution which calls for multiple VDDs can support inter-VDD communication via dynamic links.

In the case where these VDDs supplied by multiple parties, and where it is not required that all of the VDDs in the "set" be present, dynamic linking will not work.

The VDH services VDHOpenVDD, VDHRequestVDD, and VDHClose VDD are used use to overcome this limitation of dynamic linking.

Interacting with a PDD

Many VDDs virtualize hardware that generates interrupts. Generally, these VDDs will have to interact with a PDD to fully virtualize the device. VDHOpenPDD is used to establish communication between a VDD and a PDD. This is done by exchanging entry points, which use the 32-bit PLM calling convention.

From that point, the VDD and PDD are free to communicate via whatever private protocol they choose, including exchanging register-based entry points.

The VDD and PDD should also agree on a shutdown protocol that is used to stop the communication in the event the VDD needs to shutdown.

Interacting with DOS emulation

VDHAllocDosMem and VDHSetDOSDevice are used to allocate memory from the DOS memory heap and register a DOS device driver. VDDs use these services to create "fake" DOS device drivers, e.g., EMM uses this to create a stub EMM DOS device driver because DOS apps may try opening the EMM device to determine if EMM is present.

Idle DOS Application Detection

There is no standard API for a DOS application to call to inform DOS that the application is idle. The DOS application spins in a loop waiting for something to happen. When a keyboard or mouse event occurs, the DOS application takes an appropriate action, and then continues spinning.

Since DOS is a single-tasking system, this is not a problem. In a multi- tasking system like Cruiser, however, this is a total waste of CPU time.

The services VDHPostIdle, VDHWakeIdle, and VDHNotIdle allow VDDs to tell Cruiser when a DOS application appears to be idle, and when there is some activity that is like to make the DOS application busy.

For example, the keyboard VDD calls VDHPostIdle every time it gets a ROM BIOS Keyboard Peek (usually INT 16h, AH=01). The mouse can do the same thing for mouse status requests (INT 33h, ??). VDHPostIdle takes a single parameter, which is a "weight" for the amount of idleness. Cruiser adds up these weights, and if the sum exceeds a threshold (calibrated for the CPU speed), the VDM is determined to be idle, and it is put to sleep for a few time slices.

When a VDD detects that the VDM is busy, it calls VDHWakeIdle or VDHNotIdle. If the VDM was sleeping as a result of VDHPostIdle, it is immediately woken up. Otherwise, VDHWake/NotIdle do nothing.

VDHWakeIdle is used at interrupt time, when the VDD is not necessarily in the context of the target VDM. For example, the keyboard and mouse VDDs make this call when they get an event from the Shield (the VDM is running in a window) or from the corresponding PDD (the VDM is running full-screen).

VDHNotIdle is used at task time in the context of a VDM. This is a very fast call that resets the VDHPostIdle total, to prevent the VDM from being put to sleep. This call is used sparingly, at key points, to ensure that DOS applications with special behavior are not incorrectly determined to be idle. For example, the video VDD can occasionally make this call when it detects the screen has been updated. This allows an app that alternately polls the keyboard and writes to the screen to continue to run.

VDD Structure

VDDs conform to the LX executable format. A typical VDD will have a shared (global) code object, a shared (global) data object, and a private (instance) data object (*). The global code and data objects are addressable in ALL process contexts. As with instance data for DLLs, the instance data object is per-VDM data (same address in each VDM context, initialized to the same initial state, but backed by different physical memory). Each of the above three types of objects have both resident and swappable flavors. Resident objects should be used only for code and data that are accessed at hardware interrupt time.

(*) Since we use the Loader to load VDDs, a VDD may have multiple objects of each type. This is expected to be rare.

Run-time allocated memory can be allocated either swappable (Expanded Memory Manager virtual memory, for example), or fixed (interrupt simulation control blocks, for example).

Run-time allocated memory can be:

- Private (per-VDM), which is allocated out of the VDM private memory area.

Preferred method. Suitable when the memory is only used in the context of the owning VDM.

- Shared (global), which is allocated from the System area

Should be used only when the allocation must be created/modified/freed in an arbitrary process context. For example, the video VDD uses global allocations for video buffers.

VDD Loading and Initialization

VDDs are loaded at boot time, soon after PDD loading is complete.

The global code and data objects are loaded into the System area. This is required so that a PDD (PDD) can call a VDD at interrupt time to pass along a hardware interrupt, regardless of the current process context.

The instance data object(s) are loaded below 4Mb, in the private address space managed by the VDMM. The loader and VDMM conspire to make this allocation occur so that the full "virtual" state of the VDM is contained below the 4Mb line. This allows a single Page Directory Entry (PDE) to map a VDM context, which brings two benefits: 1) context switching VDMs is fast, since only a single PDE needs to be edited; 2) it is simple to maintain an alias region in the System area for each VDM.

This alias region is a 4Mb-aligned, 4Mb chunk of linear space called a "VDM Handle", or HVDM, because it can be used in a base register to manipulate the instance data of that VDM. This is a very important feature, as it allows VDDs to quickly access the instance data of ANY VDM.

After a VDD is loaded, it performs the following operations (as appropriate):

- Verify presence of corresponding hardware.
- Establish communication with corresponding PDD.
- Reserve regions of linear memory containing device ROM and RAM.

- Save initial physical device state for initializing virtual device state upon VDM creation.
- Set hooks for various VDM events (Creation, Destruction, foreground/background switch).

Unlike PDDs, VDDs are initialized at Ring 0. Hence, they cannot make Ring 3 API calls. Their only interaction with OS/2 is through the VDH services.

VDD per-VDM Initialization

Almost all VDDs will want to get called at VDM creation time to perform the following operations (as appropriate):

- Initialize virtual device state
- Initialize virtual ROM BIOS state
- Map linear regions (reserved at boot time) to physical or linear memory
- Hook I/O ports
- Enable/Disable I/O port trapping
- Hook software interrupts
- Allocate per-VDM memory

VDD communication with Kernel, PDDs, other VDDs and PM

The Dynamic Link mechanism is used to:

- Export VDH services to VDDs.
- Export kernel data to VDDs.
- Export services from one VDD to another, especially the Virtual Programmable Interrupt Controller VDD (VPIG).

To restrict the modifications required for PDDs (which remain 16:16), VDH services are provided for a VDD to establish communication with a PDD. This is no more than an exchange of entry points. At that point, the PDD and VDD may use any protocol desired to communicate. A suggested protocol is provided, but VDDs are not restricted to this.

VDD Error Handling

If a VDD calls a VDH service with an invalid parameter, it will be considered a fatal error and will result in the system being halted. This is such a serious error on the part of the VDD that it may indicate that the VDD has damaged crucial system structures. It is not considered safe to continue at this point.

VDDs invalid parameters are listed in VDH services as per the case.

The other kind of error handling may be required when a VDDs demand can't be met by a VDH service such as memory allocation or any other resource allocation. In such cases VDH services will return an error to the calling VDD and that VDD in turn will terminate the associated VDM. Rest of the MVDM will not get effected at all.

Local/Global Information Regions

Infosegs are not supported in the 32-bit world. Most of the information that was available via info segments can be obtained by a VDD using the service VDHQuerySysValue.

Architectural Review

To be performed.

VDM Manager

This page is intentionally left blank.

VDM Manager Design

+-----+	4Gb
~	~
+-----+	
VDM 2 alias	
+-----+	(System Area)
VDM 1 alias	
+-----+	
VDM 0 alias	VDM alias region
+-----+	
~	~
+-----+	512 Meg linear arena border
+-----+	4meg (end of VDM's linear region)
~	~
Per VDM memory	(Per VDM memory grows upward)
- - - - -	
~ VDD Instance data ~	
+-----+	1meg + 64K
A20 wrap area	
+-----+	1meg
CBIOS ROM	
+-----+	F0000h
~	~
+-----+	C0000h
Display memory	
+-----+	B0000h
Reserved	
+-----+	640K (A0000h) (RMSIZE/top of arena)
EBIOS data area	
+-----+	639K (start of PS/2 EBIOS data)
EMM alias region	
+-----+	256K
+-----+	64K fixed, contiguous, 64K boundary
~	~
+-----+	4K (1000h) page 0
+-----+	
DOS em code/data	
+-----+	700h
DOS comm area	
+-----+	500h
ROM data area	
+-----+	400h
Int vector table	

VDM Address Space Management

At initialization time the kernel is passed from an OEM adaptable component, a map of physical memory and ROM. This memory map describes physical page 0 and the "EBIOS" data page on a PS/2, as "ROM" data areas and are reserved for initializing the "virtual" version of this address space during VDM creation. This memory map is saved and used to map in the ROMs into the VDM address space. This is handled and described in VBIOS component. Here a brief account is listed to give the overall understanding of VDM address space.

The first 4K (page 0) contains the interrupt vector table, ROM BIOS data area, DOS communication area, etc. The VDDs maintain any state in the ROM data area.

VDDs are allowed to map physical objects like ROM, display memory or other linear memory (LIM objects, shadow display memory, etc.) in the VDM's address space. Mappings are not allowed where memory has already been allocated. Similarly memory can not be allocated where mapping already exist. Such a mapping must be unmapped before another mapping is placed at that address. Freeing linear memory invalidates the region causing page faults to be generated if the region is accessed. A partial physical mapping can not be freed, i.e. if a 64K mapping is made, the whole 64K mapping has to be freed before two 32K mappings can be put at that address.

When a VDM is created, a separate linear arena is created like any other process. The only difference is that a 4meg (PDE) linear alias is made to the VDM's context in the global (above 512meg) area. The VDM's handle is a pointer to this alias and allows VDDs to access non-current VDM contexts. The memory manager reserves an array of 4meg linear address blocks for this aliases.

After the VDM's linear arena is created, actual memory within the VDM is allocated. If this is a PS/2 machine, the 4K "EBIOS" data area is allocated and initialized from the physical region at 639K reserved at init time. The EBIOS data area is not on 4K physical page boundary.

If a VDD needs a "free" region in the VDM's memory map not at any specific linear address, the VDHQueryFreePages service is provided to search for a region where no memory has been allocated or mapped. EMM uses this service to find a 64K to 128K alias window above 640K. EMM will do this at VDM creation time instead of VDD initialization time so that other devices have a chance at reserving the ROM and device memory areas (which always should be done at VDD init time). After finding this region, VEMM will call VDHReservePages to reserve it.

The VDDs are called to allocate and map display shadow memory, conventional LIM 3.2 windows, etc. When the VDD returns to the kernel, the holes left in the DOS memory arena are allocated.

VDM Creation

The following section describes what is necessary to create a VDM. The API DOSCreateVDM is used to create a VDM. This function creates a separate VDM task, initializes it with a VDM address space and runs the DOS application.

VDM creation consists of the following steps:

1. The shell calls the kernel to create a VDM. The parameters passed to the kernel are:
 - a. Initial working directory.
 - b. CONFIG.SYS like configuration parameters (RMSIZE, FCBS, BREAK, SHELL, DEVICE, VDD configuration strings).
 - c. DOS application's full pathname and parameters.
 - d. Dos device drivers and there parameters
2. The VDM process is created. Most of the generic process creation functions are done as are applicable for protect mode process creation. VDM process creation involves:
 - a. Create and initialize a PTDA.
 - b. Allocate a process slot and pid.
 - c. Memory manager creates a new linear arena for this process.
 - d. The 4meg global PDE alias to the new linear arena is created.
 - e. Start the VDM process. The rest of VDM creation is done in this new processes context. The creating thread waits until creation is finished.
3. Call the 8086 emulation component initialization.

4. The VDM's address space is allocated and initialized as follows:
 - a. The per VDM kernel data is allocated and placed in the "per VDM memory" area (below 4meg).
 - b. The VDD instance data is allocated and placed in the "per VDM memory" area (below 4meg). See section on per VDD VDM data.
5. VDD's VDM creation entry points are called. The VDDs does as follows:
 - a. Parse the VDD configuration strings.
 - b. Allocate display shadow memory, etc.
 - c. Make aliases to physical display memory, display shadow memory or other memory mapped I/O devices, etc.
 - d. Optionally allocate memory for the 256K - RMSIZE range. The EMM VDD allocates some or all the DOS arena memory in order control the aliasing of it later. The EMM driver usually allocates a block of memory and alias it in the 256K to RMSIZE (top of VDM) range so this block can be a LIM 4.0 "alias window".
6. Finish the VDM's address space initialization by allocating memory for the holes left in the DOS arena memory.
7. Call the DOS emulation component initialization. It does the following:
 - a. Initialize the DOS arena.
 - b. Load DOS device drivers.
 - c. Execute shell specified in SHELL.

VDM Destruction

When the top process invoked by the user in a VDM terminates or the VDM is abnormally terminated because of some illegal action, the entire VDM disappears immediately.

Here are the following steps taken during VDM termination:

1. The DOS application terminates normally or abnormally.
2. The VDD's VDM termination entry points are called.
3. The regular process termination code is called in the context of the dying process, and file system, semaphores, memory manager, etc. resources are cleaned up, and process actually dies.

Per-VDM VDD Data Management

This section describes how VDD handle their per VDM data (instance data). The VDD contains a separate linear region called an "instance region" much like instance segments in dynlink libraries. All references in the VDD to this region are fixed up to a region in the VDM between 1meg + 64K and 4meg with the same linear address for all VDM processes. A base register isn't needed to access variables in the current VDM context:

```
mov     eax, [InstanceVar]
```

For code that must run in non-current VDM contexts:

```
mov     eax, [ebx].InstanceVar
```

It is important to note that instance data is NOT directly accessible at interrupt time. A VDD accessing this data must use the appropriate VDM alias [HVDM]. This is required because instance data is in per-VDM private address space.

Example of task-time instance data usage:

```
mov     ax,myInstanceWord           ; ax = instance word
```

Example of interrupt-time instance data usage:

```
mov     ebx,VDMGlobalInfo
mov     ecx,[ebx].vgi_hvdmForeground ; ecx = foreground hvdm
cmp     ecx,HVDM_BAD                ; Any VDM foreground?
jz      xx1                          ; NO, skip work
mov     ax,[ecx].myInstanceWord      ; ax = instance word
```

VDM/VDD Page Fault Services

For the video VDD's display memory "copy on demand" or "lazy" copy support, a mechanism is required to route VDM page faults to VDDs. If a page fault is received on an invalid page in a VDM context, the fault is passed to the VDM support component. Based on the faulting page address and current VDM context, the fault is routed to the appropriate VDD. If there is no VDDs registered for this region, the faulting region is mapped to don't care pages. Don't care pages are provided to this component by IBMBIO.COM at init time.

The "fault region" start, size and VDD fault handler address is kept in a table setup via the VDHInstallFaultHook VDD service. This can only be done to a region initialized as an alias. When a page fault occurs in the VDM address space, this table is searched for a matching region and the VDD handler address is called. The faulting page address and the kind of fault that occurred is passed to the VDD handler. When the VDD doesn't want to fault on these "fault regions", it can map them to don't care regions. The fault handlers are only called when a page in region is invalid.

Memory management services

These services allow VDD's to allocate/free/reallocate memory with different options such as fixed/swappable etc. At VDM creation time VDM creates and initializes a linear heap for each VDM. This is represented in the form of a linked list and each record stores a linear region with its different attributes i.e. reserved/allocated/mapped and page/byte granular. Allocation of memory is always page granular but byte granular services are provided from a byte heap with initial allocation of one page. In case of growing need this byte heap is reallocated with a bigger size. This byte granular services are required for instance data reservations and allocation of memory in dos arena. Four types of mappings are supported: Mapping to a physical arena, mapping to another linear arena, mapping to don't care pages and mapping to invalid pages which means unmapped. These mappings change the status of the linear heap record and then request is passed on to the VM to actually change the mappings. VDD's can also request for small memory allocation and freeing from kernel heap which is global and fixed. Small, fixed size block services are also available to speed up the frequent allocations and freeing of memory particularly for VPIC. For a particular block size a pool of blocks are maintained and the requirements are met by taking off a block from the block pool.

Semaphore services

These are intended for synchronizing with an OS/2 process. VDDs must be careful not to block (VDHRequestSem/VDHWaitSem) in the context of a VDM task, or that VDM task will receive no more simulated hardware interrupts until it becomes unblocked.

Timer/Priority services

Timer services are required by VPIC for IRET time outs and VVIDEO for screen switching to pass on the control to a handler after a certain amount of time. Timers implanted can be deregistered before they expire. Priority services are provided to set VDM's scheduler priority used by VPIC to schedule the pending interrupts fast.

Freeze/Thaw services

These services are basically used by video VDD. A freeze service will freeze the operation of a VDM and thaw will put it again on the scheduler queue. It also ensures that a VDM will come out of its freeze state only when number of thaw request made are equal to the freeze requests.

Imported Interfaces

- Em86CreateVDM	For 8086Em initialization in VDM creation for
- DemStart	For Initialization of DOS emulation
- TKProcessCreate	Create a new process indicating for a VDM
- TKProcessKill	Kill a process
- VMAlloc	Allocate a memory object
- VMFree	Free a memory object
- VMReAlloc	Reallocate a memory object
- alloc	Allocate a block from resident heap
- free	Free a block to resident heap
- salloc	Allocate a block from swapable heap
- sfree	Free a block to swapable heap
- VMMapAlias	Map virtual to virtual/physical
- VMQueryMaxAlloc	Get maximum swapable pages present
- w_TimerStart	
- w_TimerStop	
- w_SemSet	Set semaphore
- w_SemClear	Clear semaphore
- w_SemWait	Wait on semaphore
- w_SemRequest	Request to set a semaphore
- w_SetWait	Set and wait on a semaphore

Internal Interfaces

Design Constraints

- The "global" data areas in the ROM BIOS areas are relatively small (10-20 bytes).
 - CTRL-ALT-DEL is not virtualized for VDM's. It has the same effect as when done when a protect mode application has the focus.
-

Design Review

To be performed.

VDM Manager Implementation

This page is intentionally left blank.

Internal Interfaces

To be added.

Implementation

To be added.

Implementation Review

To be performed.

VDM Manager Appendix (included for each review)

This page is intentionally left blank.

Glossary

To be added.

.....

- ation name). This function creates a process.
- to create the VDM process. There is no need to create a separate process for each thread. The threads are created and treated separately from the process.
- environment is not available if a separate process is created.
- ation name). This function turns the process into a thread. The Presentation Manager will create a separate process for each thread.
- environment is available for VDM threads. This is easier to create separate threads for each thread.
- ated and treated like any other process. Less code to implement a thread.
- link instance data, LDT overhead, etc.

- called an "instance region" much
ed up to a region in the VDM betw
ister isn't needed to access varia

contexts:

nceVar

- gister.

- Requires special loader support. This component provides the loader with a function that reserved linear address space in the 1 to 4 meg region for the VDD instance data.
- 2. During VDD initialization, the VDD informs the kernel how much instance data is needed and the kernel returns the instance data's offset from the VDM's base linear address (the VDM handle). To access an instance data item, the VDM handle is added to the offset returned from the call at init time plus the offset of the data item within the data area:

```
add    ebx,[VDD_VDM_DataOffset] ; setup at init time
mov    eax,[ebx].InstanceVar
```

If the code is always executed in the current VDM context, this is a very minor optimization:

```
mov    ebx,[VDD_VDM_DataOffset] ; setup at init time
mov    eax,[ebx].InstanceVar
```

- + Doesn't require special loader support for instance segments.
- + Follows the Windows/386 model of VDM data; makes adapting their code easier.
- Slower and wastes a register.

Windowed VDM

This page is intentionally left blank.

Windowed VDM Architecture

This page is intentionally left blank.

Problem Description/Objectives

Cruiser currently provides only full-screen support for Multiple Virtual DOS Machine sessions (MVDM). This DCR addresses the impacts and problems related to the windowing of DOS applications under Presentation Manager for the Cruiser release. The following affected areas have been identified, and are discussed in detail below:

- VDM Manager
- VDH Services
- Video VDDs (VCGA/VEGA/VVGA)

- Keyboard VDD (VKBD)
 - Mouse VDD (VMOUSE)
 - Shield (PMVIOP/PMVDMP)
 - Graphics Engine (PMGRE)
 - Window Manager (PMWIN)
 - Base Session Management (SESMGR)
 - PM Session Management (OS2SM)
 - Shell (PMSHELL)
 - Command-line Support (CMD)
 - Translation and Code-Page Considerations
-

Solutions/Justification

To compete effectively against other multitasking and windowing environments for DOS applications (e.g., Windows/386, DesqView), and to provide a high degree of seamlessness between DOS applications and other classes of windowable applications under OS/2 (eg, PM, VIO, Posix), it is important that we support the operation and management of DOS sessions inside PM windows. This support is likely to play a significant role in broadening the end-user acceptance of OS/2, because it will eliminate most of the advantages of other PC multi-tasking environments over OS/2.

Windowed VDM Design

This page is intentionally left blank.

Overall Objectives

- Display and operate text-mode VDMs in windows
- Display graphics-mode VDMs in windows

NOTE: Graphics applications may be "frozen", i.e., inoperable, when running in a window, if the Video VDD chooses not to support a particular graphics mode.

- Provide windowed performance better than Windows/386
- Provide user interface consistent with Windowed VIO sessions

With regard to windowed performance, we have the following specific goals:

- TTY output must be transferred to a VDM window quickly
- Direct video memory accesses should be isolated "intelligently"

TTY output includes common console output operations, such as DIR or TYPE, and any application using the INT 10h WriteTTY and/or WriteString services to access the display.

"Intelligently" means choosing appropriate times to check an LVB for changes. We have defined specific mechanisms, which include a periodic timer, video software interrupt interception, and keyboard input notification, that will allow the VDM windowing implementation to be responsive and fast. We will also have the option of fine-tuning LVB updates to focus on areas that are likely to change most often, based on either a static description of typical LVB accesses, or a dynamic record of actual LVB accesses; however, those and other performance-tuning options will not be discussed within this DCR.

Overall Design

There will still be a limit of 16 simultaneous Virtual DOS Machines (VDMs); we will simply allow the user to convert full-screen VDM sessions to windowed VDM sessions and vice versa. The system menu for VDM windows will be augmented with all the options currently available to VIO windows, as well as a new option to change the default viewing mode (windowed or full-screen), and a special key combination (Alt-Enter) for quick toggling between viewing modes. This capability is a necessity for VDMs, since we cannot guarantee that a VDM that began life windowable will remain windowable (ie, it may use a graphics or other video mode that cannot be virtualized).

[Issue: Windows/386 uses settings in WIN.INI to determine whether special key combinations should be passed to VDMs or not]

The restrictions on supported VDM video modes will be the same as before (that is, full-screen VDMs are unrestricted, whereas background VDMs can run only as long as they remain in a text mode). Windowed VDMs will have the same restrictions as background VDMs. As long as a VDM window remains in text mode, its contents, operation, and overall appearance will be identical to that of a VIO window. Video updates in the window will be performed based on a combination of events in the VDM (eg, INT 10h, INT 16h) and a periodic timer. Updates will be synchronized with scroll operations to provide reasonably smooth output. The Shield (PMVIOP) will communicate with the Video VDD to obtain video update information, with the Keyboard VDD to provide virtual keystrokes and obtain changes in VDM shift states, and with the Mouse VDD to provide mouse input to the VDM.

User Interface

Installed DOS applications have one of the following "how-to-run" attributes:

1. Run the program full-screen
2. Run the program in a text window

Unlike VIO windows, there is no option to "let the program decide", because DOS executables do not provide the necessary information. Also unlike VIO windows, this attribute is dynamic, and can be changed at any time; the value specified and retained by the Desktop Manager is simply the initial value of the attribute.

Installed DOS applications also have one of the following "how-to-close" attributes:

1. Close the window on program exit
2. Retain the window on program exit until closed by the user

The treatment of this attribute is identical to that for VIO windows. When the first DOS application to be loaded in the VDM exits, in the first case the window is immediately destroyed, and in the second case it is preserved (but flagged as "Completed" in its title bar) until an explicit close by the user.

Full-screen VDM sessions appear on the PM desktop as icons only, and have the following options on their system menu:

- Restore Restore VDM to full-screen
- Move Move VDM icon
- Close Close VDM session, terminating all its DOS apps

- Switch To (same as Ctrl+Esc)
- Help (similar to VIO Window help)
- Next Window (same as Alt+Esc)
- Windowed Change VDM to windowed, and perform implicit restore
- Copy All Copy the entire VDM screen to Clipboard
- Paste Transfer Clipboard contents to window as input

OS/2 full-screen sessions.

Windowed VDM sessions exist on the PM desktop as either icons or windows. When as icons, they have the following system menu options:

- Restore Restore VDM icon to window
- Move Move VDM icon
- Maximize Restore VDM icon to maximum window
- Full-Screen Change VDM to full-screen, and perform implicit restore
- Close Close VDM session, terminating all DOS apps therein
- Switch To... (same as Ctrl+Esc)
- Help... (similar to VIO Window help)
- Next Window (same as Alt+Esc)
- S/L Font/Pixel Display using small/large font (or pixels if graphics)
- Scroll DISABLED
- Mark DISABLED
- Copy DISABLED
- Copy All Copy the entire VDM screen to Clipboard
- Paste Transfer Clipboard contents to window as input

And when displayed as a window, a windowed VDM session has these system menu options (same as above except for Restore, Move, Size and Minimize):

- Restore DISABLED
- Move Move VDM window
- Size Resize VDM window
- Minimize Shrink to VDM icon
- Maximize Restore VDM icon to maximum window
- Full-Screen Change VDM to full-screen, and perform implicit restore
- Close Close VDM session, terminating all DOS apps therein
- Switch To... (same as Ctrl+Esc)
- Help... (similar to VIO Window help)
- Next Window (same as Alt+Esc)
- S/L Font/Pixel Display using small/large font (or pixels if graphics)
- Scroll Enter/exit scroll mode (arrows trapped to scroll window)
- Mark Enter/exit marking mode
- Copy Copy the currently marked portion to the PM Clipboard

- **Copy All** Copy the entire VDM screen to Clipboard
- **Paste** Transfer Clipboard contents to window as input

All options shown as DISABLED will be listed but unselectable (grayed). Copy will be selectable only once an area has been marked. Choosing Copy will also cancel marking mode and any marked area. Paste will be selectable only if there is text currently in the Clipboard.

When a windowed VDM switches to a graphics mode, it will be frozen and flagged as "Suspended" in its title bar. Regardless of whether a VDM is frozen or not, the VDM window can be manipulated in most of the usual ways. Specifically, it can be moved, resized, minimized, restored, maximized, scrolled, closed, marked, copied, or redisplayed with a different font or pixel size. The only restrictions for VDM windows that do not also exist for VIO windows is:

- Limited paste capability when VDM is frozen (constrained by the size of the Keyboard VDD's scan-code buffer)

Conversely, VDM windows will lack some of the restrictions that exist for VIO windows; in particular, they will have:

- Private keyboard shift states
- Ability to display graphical applications (albeit, frozen)

Since VDMs can select different graphics resolutions, and since those resolutions will often be smaller than the resolution used by PM, an option unique to VDM windows will be provided that allows the user to display the window contents at either PM's resolution or at the resolution closest to the VDM's. This is what's referred to as the Small/Large Pixel option (it replaces the Small/Large Font option when a VDM window is in graphics mode). If for some reason the installed PM Display Driver has a lower maximum resolution than that selected by a windowed VDM, the user will simply be unable to maximize the window in such a way that it is completely visible at once.

Small/Large Pixel support does not require any additional Graphics Engine services. GreBitBlt can be used to stretch a bitmap by providing a target rectangle that is larger than the source.

VIO vs. VDM Window Management

The Shield (PMVIOP) manages windowed VIO session input/output as follows:

```
+-----+
|Keyboard input|
| Kbd -> SINGLEQ -> PMWIN -> PMVIOP -> BKSCALLS -> KBDCALLS -> App |
|Mouse input  |
| Mse -> SINGLEQ -> PMWIN -> PMVIOP -> BMSCALLS -> MOUCALLS -> App |
|Screen output|
| App -> VIOCALLS -> BVSCALLS -> VDHWNDW -> PMGRE -> DISPLAY -> Screen |
+-----+
```

Input/Output Flows for Windowed VIO Sessions

For windowed VDMs, the Shield will manage input/output in a similar fashion:

```
+-----+
|Keyboard input|
| Kbd -> SINGLEQ -> PMWIN -> PMVIOP -> VKBD -> DOS App |
|Mouse input  |
| Mse -> SINGLEQ -> PMWIN -> PMVIOP -> VMSE -> DOS App |
|Screen output|
| DOS App -> VVIDEO -> PMVIOP -> PMGRE -> DISPLAY -> Screen |
+-----+
```

Input/Output Flows for Windowed VDM Sessions

For VIO applications, VIO calls are shadowed in a logical video buffer and translated by a special dynlink (VDHWNDW) into appropriate PM Graphics Engine (PMGRE) calls; thus, the window is updated synchronously and in the context of the caller. Both the Shield and the VIO subsystems have direct access to the session's logical video buffer (LVB) and other shared data concerning the state of the session. The shared data is protected by semaphores, so that while the Shield is accessing it (ie, to repaint the window after a move or resize), the application is locked out, and vice versa.

For windowed VDM output, the Shield must play a much larger role. In this case, the Video VDD will intercept all I/O instructions to the video adapter, and all screen updates will be redirected (ie, mapped) to a logical video buffer maintained by the Video VDD. When it detects

changes in a VDM's video state, it notifies a new Shield thread, providing it with update information for a specified VDM window. This thread is called the "VDM Event Thread".

The VDM Event Thread will be a high-priority thread that serves the needs of all windowed VDMs; more threads would increase system overhead without substantial gain, since unlike VIO windows, a context switch is required for any VDM-initiated window updates, and calls to the Graphics Engine are (for most video hardware) fully CPU-bound.

For keyboard and mouse input, the Shield must simply transform keystroke and mouse event messages into calls to the Keyboard and Mouse VDDs. Unlike VIO windows, no buffering by the Shield is required, because the Keyboard VDD maintains its own virtual scan-code buffer for every VDM. The Mouse VDD will be similarly modified to buffer events, eliminating the need for buffering not only in the Shield but in the Physical Mouse Device Driver (Mouse PDD) as well. The rationale is that ring transitions from Shield to Mouse VDD (to post events) are preferable to context-switches from Mouse VDD to Shield (to request events).

The mechanisms used by the Shield to communicate with the Video, Keyboard and Mouse VDDs are Dos32OpenVDD, Dos32RequestVDD, and Dos32CloseVDD. The specifics of all the services available through Dos32RequestVDD are discussed below, under the sections on VDM Video, Keyboard and Mouse Support.

Virtual Display Management

The Shield is notified by the Video VDD of the following changes, which are prioritized in the order shown (from high to low):

1. Change in video mode (including code page change and freeze/thaw)
2. Change in palette
3. Change in LVB
4. Scroll of LVB
5. String output
6. Change in cursor
7. Input notification (used to adjust window scroll based on cursor position)
8. Paste notification (i.e., VKBD is ready for additional paste data)

Having received notification for one of these changes, it is the Shield's responsibility to make appropriate changes to the VDM window, usually through the use of one or more PM Graphics Engine call(s).

Although well-defined video interfaces exist within VDMs, such as INT 10h, DOS applications often write to VRAM directly. And although each direct read/write operation could be detected by the Video VDD using page faults, the performance would be unacceptable, and the determination of the precise type and size of each operation extremely difficult. So the Video VDD treats a windowed VDM just like a background VDM, by mapping the VDM's LVB into its VRAM address space. The VDD uses a 16Hz timer to periodically check for dirty pages in the VRAM address space of every windowed VDM. When a windowed VDM is found with dirty pages, the VDM Event Thread is notified of an LVB change, along with rectangles describing which areas of the VDM's LVB may be dirty. The Shield finds the smallest rectangle(s) of change, and updates the window using appropriate Graphics Engine services.

Although the Video VDD manages all of a VDM's virtual VRAM, both visible (on-screen) and non-visible (off-screen), as far as the Shield is concerned the LVB is strictly the visible portion of that memory. So when a VDM changes its active video page (also called video page flipping), the flip will be handled as a complete LVB update. The rectangles passed to the Shield will always be adjusted to represent just the visible portion, and will be set by the Video VDD automatically in the event of a video page flip; in other words, even though the application may not have modified any virtual VRAM, page flipping will appear to the Shield as a total LVB modification.

Not all changes in the LVB are reported at 16Hz intervals. When a VDM makes INT 10h WriteString calls, the Video VDD will report the exact string location and length to the Shield immediately. When a VDM makes INT 10h WriteTTY calls, the VDD will coalesce them until a CR or LF is detected, and then report single string and cursor events. As safeguards, the VDD will always report what it has received thus far on the next timer tick and/or after some amount of characters has been received (to be defined). Scrolls and unnested cursor movements are also reported immediately; "unnested" means that the cursor was moved explicitly, rather than implicitly as the result of a WriteTTY or WriteString call.

The Shield obtains access to the VDM's LVB indirectly, by requesting the Video VDD to copy some rectangular portion of it into a "shield buffer". The Shield compares the "shield buffer" against a "shadow buffer", which contains the previously displayed contents of the VDM window, to find the smallest areas of change. The roles of the two buffers are then interchanged in preparation for the next update, as it is now the "shield buffer" which contains the last data displayed. See the ALTERNATIVES section below for alternate approaches.

If the VDM changes video modes before the Shield is able to copy the VDM's LVB, an error will be returned to the Shield on the copy request. The Shield's action will be to query the new mode and recopy the LVB.

Virtual Keyboard Management

The handling of keyboard input is considerably less work for the Shield for VDM windows than it is for VIO windows. The Shield simply transforms keystroke messages into calls to the Keyboard VDD. No buffering by the Shield is necessary, since the Keyboard VDD already maintains its own virtual keyboard buffer.

The only keystrokes that require special handling by the Shield are those that affect shift states. As VDM windows gain focus, the Shield must pass "manufactured" make and/or break events to them if their shift state at the time of the last event posted to them is now out of sync with the current physical shift state.

The set of operations that a VDM can perform on its virtual keyboard includes:

1. Change repeat rate
2. Change shift states

When a VDM changes the repeat rate, whether foreground, background or windowed at the time, the repeat rate is changed for the whole system. Since repeat rate is not a per-session attribute under OS/2, the Shield need not be involved. Shift state info (and the LEDs for focus VDMs) are per-VDM and are managed by VKBD, in conjunction with the physical keyboard driver.

Virtual Mouse Management

For mouse input, the Shield will transform mouse event messages into calls to the Mouse VDD. The conversion will require mapping the physical mouse position reported by PM to a corresponding position within the VDM's screen dimensions, offset appropriately by the window's origin within the VDM's virtual screen. Any events that fall outside the VDM's virtual mouse grid (if an alternate grid is even defined) must be discarded by the Mouse VDD, as we are not architecting any interface between Shield and Mouse to describe, or notify of changes in, virtual mouse grids.

The VDM can change the state of its virtual pointer in various ways:

1. Change pointer position
2. Change pointer shape
3. Change pointer scaling factors

but as with VIO windows, none of these changes are of any interest to the Shield. An application can move its virtual pointer wherever it pleases, but that position will be changed by the Shield as soon as the user generates a mouse event inside the VDM window. The pointer shape selected by a VDM is also unimportant, because when a VDM is windowed, the Video VDD will disable pointer-drawing; as is the case for VIO windows, PM's pointer always remains visible.

Unfortunately, there are some apps (QuickBasic?) that draw their own pointer. However, this is really only a nuisance, not a serious technical problem, and we will not do anything to address it. In the context of text-mode-only applications, this is not expected to be a common problem. We have taken the same position for VIO sessions; namely, that while VIO applications could draw their own pointers, since most do not, we can ignore the usability problems that would arise from having two pointers (PM's and the application's) visible in the same window.

Like VIO windows, VDM windows will be scrollable by dragging the mouse from the inside of the window to the outside. Each time the window is scrolled some small amount, another mouse event will be generated at the bottom of the window, but relative to the new window origin. Since a button must be held down to perform this action, the Shield will take care to send a corresponding button-up event to the same VDM.

Design Review

To be performed.

Windowed VDM Implementation

This page is intentionally left blank.

Internal Interfaces

To be added.

Implementation

To be added.

Implementation Review

To be performed.

Windowed VDM Appendix (included for each review)

This page is intentionally left blank.

Glossary

To be added.

Size and Performance Considerations

To be added.

Implementation Estimates

To be added.

Automatic screen-switching

When a windowed text VDM switches to graphics mode, we could provide the user with an option to automatically switch such a VDM to full-screen mode, so that it could continue to run. Similarly, when such a VDM reset back to text mode, we could automatically screen-switch back to PM and redisplay the VDM in a window. However, to avoid cluttering the user interface, and to save a small amount of work for a feature that is likely to be little- used, it is recommended that such windows be listed simply as "Suspended" in their title bar, and that the user be required to explicitly control their state.

LVB Update Processing

To reduce window update time and virtual interrupt dispatch latency, there are several alternate approaches to LVB update processing. One is to have the Video VDDs determine where LVB changes have been made, rather than the Shield. A Video VDD would do this by maintaining a shadow LVB and periodically comparing it against the VDM's actual LVB, then recording where the changes occurred and updating just that portion of the shadow LVB. Shield notification would then include the location and size of the area(s) changed, and an alias to the Video VDD's shadow LVB. This is very similar to the proposed solution (where the Shield rather than the Video VDD maintains the shadow buffer), but has the following advantages:

- The time a VDM was frozen would be slightly shorter because the freeze would not have to remain in effect across a context-switch to the Shield and one or more Shield requests to the Video VDD
- By doing the comparison before the shadow buffer update, only the modified area(s) would need to be transferred to the shadow buffer afterward

and the following disadvantages:

- Requires more services (to alloc/free page or selector-based aliases)
- Requires every Video VDD to provide a common piece of code that is largely device-independent

There are other possible combinations. For instance, the Video VDD could simply provide the Shield with an alias to a VDM's LVB and a dirty video bit vector, and let the Shield maintain, compare, and selectively update its own shadow buffer. However, the necessary freeze duration would not be shortened; we would simply be saving the difference in time between always copying 4Kb (or 8Kb) to the Shield's buffer, and copying only what had changed to the shadow buffer.

Note that Windows/386 makes no attempt to selectively update their shadow buffers; once a shadow page has served its purpose in pinpointing an area of change, it is automatically updated with a quick single 4K copy. So the real issue here is minimizing the amount of time a VDM is frozen, and assuming that the VDM Event Thread dedicated to handle VDM repaints is high priority, it would be uncommon for the VDM to be scheduled before the time we expect a freeze to be lifted anyway.

Advanced Properties

1. Virtual BIOS Device Driver
2. Virtual Programmable Interrupt Controller
3. Expanded Memory (LIM) Manager
4. Extended Memory (XMS) Manager
5. Virtual DMA Device Driver
6. Virtual Floppy Device Driver
7. Virtual Disk Device Driver
8. Virtual LPT Device Driver
9. Virtual CMOS Support

- 10. Virtual Timer Device Driver
- 11. Virtual Video Device Driver
- 12. Virtual Keyboard Device Driver
- 13. Idle Detection Component
- 14. Virtual Mouse Device Driver
- 15. Virtual COM Device Driver
- 16. 387 Coprocessor Support

Dos APIs

This page is intentionally left blank.

DosStartSession - Start a Session

DosStartSession allows an application to start another session and to specify the name of the program to be started in that session. New sessions may only be started in the foreground when the caller's session (or one of the caller's descendant sessions) is currently executing in the foreground. The foreground session for windowed applications is the session of the application which owns the window focus. The new session will appear in the Shell switch list.

Any protect mode application may start any other protect mode application in a new session, regardless of the issuing programs session type.

Parameters

StartData (@OTHER)

StartData is a structure containing the data describing the session to be started.

SIZE	DESCRIPTION
----	-----
WORD	Length
WORD	Related
WORD	FgBg
WORD	TraceOpt
DWORD	PgmTitle
DWORD	PgmName
DWORD	PgmInputs
DWORD	TermQ
DWORD	Environment
WORD	InheritOpt
WORD	SessionType
DWORD	IconFile
DWORD	PgmHandle
WORD	PgmControl
WORD	InitXPos
WORD	InitYPos

WORD	InitXSize
WORD	InitYSize
WORD	Reserved
DWORD	ObjectBuffer
DWORD	ObjectBuffLen

Length is the length of the data structure in bytes including **Length** itself. **Length** can be 24, 30, 32, 50, or 60 bytes for v 2.0, 24, 30, 32 or 50 bytes for v 1.1 or 24 bytes for CP/DOS v 1.0. A length of at least 32 bytes must be specified for CP/DOS v 2.0 to start a Virtual Dos Machine (VDM) session. A length greater than 32 will not be allowed if the session manager detects that the Presentation Manager is not present.

Related specifies whether the session created is related to the calling session.

- Value = 0, new session is an independent session (not related)
- Value = 1, new session is a child session (related)

An independent session can't be controlled by the calling program. It may not be specified as the target of a DosSelectSession, DosSetSession, or DosStopSession. The **TermQ** parameter is ignored for independent sessions, and **SessID** and **PID** are not returned. Refer to "**Parent/Child Relationship**" in the **Remarks** section for additional information about related sessions.

FgBg specifies whether the new session should be started in the foreground or background. If a windowed session is started in the foreground, the new session will be given the window focus.

- Value = 0, start session in foreground
- Value = 1, start session in background

TraceOpt specifies whether the program started in the new session should be exec'ed under conditions for tracing.

- Value = 0, no trace
- Value = 1, trace (No notification of descendants)
- Value = 2, trace all descendant sessions

Related equal 1 and a termination queue must be supplied when a TraceOpt of 2 is specified. Refer to "**DebuggerConsiderations**" in the **Remarks** section for additional information about a TraceOpt of 2.

PgmTitle is the address of an ASCIIZ string containing the program title. The string can be up to 61 bytes long including the terminating byte of zero. If the address specified is zero, or if the ASCIIZ string is null, the initial title is **PgmName** minus any leading drive and path information.

PgmName is either zero or the address of an ASCIIZ string containing the fully-qualified drive, path, and filename of the program to be loaded. Refer to "**PgmName/PgmInputs Considerations**" in the **Remarks** section for additional information about a zero **PgmName** address.

PgmInputs is either zero or the address of an ASCIIZ string containing the input arguments to be passed to the program.

TermQ is either zero or the address of an ASCIIZ string containing the fully-qualified path and filename of an CP/DOS queue (Reference DosCreateQueue). Refer to "**Parent/Child Termination**" in the **Remarks** section for additional information about the **TermQ** parameter.

Environment is either zero or the address to an environment string (Reference DosExecPgm) to be passed to the program started in the new session. The **Environment** may be used for independent or related DosStartSession calls. When the **Environment** parameter is zero, the program in the new session will inherit the environment of the Shell if the **InheritOpt** is equal to zero or the environment of the program issuing the DosStartSession if the **InheritOpt** is equal to one.

Interpretation of the "Environment" field for a VDM

When starting a DOS session, the "Environment" field is interpreted as as pointer to a PROPERTYBUFFER". This buffer consists of a ULONG length, followed by one or more triples of the form <property type, property name, property value>.

The OS/2 supplied properties are listed in the following table:

	Bounding Info	Class	VDD	Type	Min	Max	Step	English	Property Name	
Std	VBIOS VDMP_BOOL			"Break"	Std	VBIOS VDMP_SHELL		"Shell"	Std	VBIOS VDMP_INT 128 640 16
"DOS memory size"	Std	VBIOS VDMP_INT	0	254	1	"FCB count"	Std	VBIOS VDMP_INT	0	254 1 "FCB count, no close"
Std	VBIOS VDMP_MLSTR			"DOS device drivers"	Std	VKBD VDMP_INT	0	100	1	"Keyboard idle threshold" Ins
VEMM VDMP_INT	0	32767	64	"LIM memory maximum"	Ins	VXMS VDMP_INT	0	32767	64	"XMS memory maximum"

where

Class = standard - available on all 386 OS/2 systems. installable - available only if corresponding VDD is installed.
VDD = Virtual Device Driver responsible for registering property.

Type = Property type (see VPTYPE equates) VDMP_BOOL = boolean (true/false) VDMP_INT = unsigned integer
The min, max, and set fields restrict the values that the property can assume, and
are used to present a nice user interface. VDMP_MLSTR = multi-line string

The following MASM source is an example of a valid PROPERTYBUFFER:

```

first  label  byte
      dd      last-first          ; size of buffer, include length

      dw      VDMP_BOOL          ; type
      db      'Break',0          ; name
      dd      0                  ; value - false

      dw      VDMP_INT           ; type
      db      'DOS memory size',0 ; name
      dd      640                ; value

      dw      VDMP_INT           ; type
      db      'FCB count',0      ; name
      dd      16                 ; value

      dw      VDMP_INT           ; type
      db      'FCB count, no close',0 ; name
      dd      18                 ; value

      dw      VDMP_MLSTR         ; type
      db      'DOS device drivers',0 ; name
      db      'vt52.sys',0Ah     ; value
      db      'edit.sys',0Ah
      db      'fast.sys',0
last   label  byte

```

Handling errors in a PROPERTYBUFFER

The following errors are non-fatal, and treated in the manner prescribed. This makes the system more robust and more forgiving in situations where the set of registered properties change over time.

- Unknown property name

The property type is used to skip the property value. The offending property is ignored.

- Invalid property value

The default property value is used.

The following errors are fatal, and cause DosStartSession to fail.

- Buffer too small
- Buffer improperly formatted

This occurs when an property type/name/value extends past the end of the stated buffer size.

NOTE

A VDM **always** starts with an empty environment, regardless of the the InheritOpt setting. This is sensible because environment variables for an OS/2 process are not likely to be sensible for a DOS program. The VDM's shell must define the initial environment (COMSPEC= and PATH=). The VDM's shell will **always** process the AUTOEXEC.BAT in addition to executing the given COM, EXE, or BAT file. This allows the AUTOEXEC.BAT to set up the the environment for the VDM.

InheritOpt specifies whether the program started in the new session should inherit the calling program's environment and open file handles.

- Value = 0, inherit the Shell's environment.
- Value = 1, inherit the environment of the program issuing the DosStartSession call.

For a VDM, the DOS program inherits *only* the current drive and directory. As discussed above, a VDM always starts with an empty environment.

The **InheritOpt** may be used for independent or related DosStartSession calls. Therefore, a DosStartSession call with the **InheritOpt** equal to one is the equivalent of a DosExecPgm, except the new program doesn't inherit the priority of the parent process or the keyboard and video characteristics associated with the parent session. Also, a parent process/child process relationship is not established. Refer to "**Parent/Child Relationship**" in the **Remarks** section for additional information about related sessions. VDM Sessions cannot inherit their parent's environment. So, an InheritOpt value of 1 not allowed for starting VDM Sessions.

SessionType defines the type of session that should be created for this program.

- Value = 0, use the PgmHandle data or allow the Shell to establish the session type.
- Value = 1, start in a full screen session
- Value = 2, start in a windowed session for programs using the Base Video Subsystem.
- Value = 3, start in a windowed session for programs using the Presentation Manager services (including AVIO calls).
- Value = 4, start the program in a Virtual Dos Machine (VDM) Session.

IconFile is either zero or the address of an ASCII string containing the fully-qualified drive, path, and filename of an icon definition. The system will provide a default icon for windowed applications if an icon filename is not provided on the DosStartSession call. An icon definition filename may be specified for full screen or windowed sessions.

PgmHandle is either zero or the program handle returned by the WinAddProgram or WinQueryProgramHandle calls. The program handle identifies the program in the installation file to be started, the program title, the session type, and the initial window size and position. However, information may be specified on the DOSStartSession call to override the information in the installation file for this invocation of the program.

DosStartSession does not support program groups.

PgmControl may be used to specify the initial state for a windowed application. This parameter is ignored for full screen sessions.

This parameter is bit mapped as follows:

BIT	INITIAL WINDOW STATE
0	= Invisible
1	= Maximize
2	= Minimize
3	= No Auto Close
4 - 14	= Reserved
15	= Use specified size : position

NOTE:The 'No Auto Close' bit is used only for VIO Windowable applications, and is ignored for all other types of applications.

InitXPos and InitYPos contain the initial x and y coordinates in pels for the initial session window. Coordinates (0,0) indicate the bottom left corner of the display. This parameter is ignored for full screen sessions.

InitXSize and InitYSize contain the initial x and y extent in pels for the initial session window. This parameter is ignored for full screen sessions.

Reserved is a WORD of zeros, reserved for future use.

ObjectBuffer is where the name of the object that contributed to the failure of DosExecPgm is returned. DosExecPgm is called by DosStartSession to start all Full Screen, VIO Windowed, and Presentation Manager programs.

ObjectBuffLen is the length in bytes, of the buffer described by **ObjectBuffer**

SessID is the session ID associated with the child session created. **SessID** is returned only when the value specified for **Related** is 1. The **SessID** returned can be specified on subsequent calls to DosSelectSession, DosSetSession, and DosStopSession.

PID (@DWORD)

PID is the process ID associated with the child process created. **PID** is returned only when the value specified for **Related** is 1. The **PID** returned may **not** be used on any CP/DOS calls, e.g., DosSetPrty, which require a parent process/child process relationship. Read the paragraph with the heading "**Parent/Child Relationship**" in the **Remarks** section.

Returns

- No Error
- ERROR_SMG_INVALID_CALL
- ERROR_SMG_INVALID_SESSION_ID
- ERROR_SMG_PROCESS_NOT_PARENT
- ERROR_SMG_RETRY_SUB_ALLOC

Remarks

When a **Length** of 24 or 30 bytes is specified, DOSStartSession will initialize the missing parameters to 0. This will allow the Shell to provide values for the missing information based on the installation file entry for the program being started.

Foreground/Background Considerations:

DosStartSession will only start a new session in the foreground if the program issuing the DosStartSession or a descendent session is executing in the foreground session. Otherwise, DosStartSession will override the foreground request and start the new session in the background. A unique error is returned in AX indicating the new session was started in the background. The foreground session for windowed applications is the session of the application which owns the window focus. Therefore, when a windowed session is started in the foreground, the new session will be given the window focus.

Parent/Child Relationship:

When **Related** equal 1 is specified, DosStartSession establishes a parent session/child session relationship. A parent process/child process relationship is not established. The parent process is the shell process just as if the operator had started the program from the shell menu. Therefore, the **PID** returned by DosStartSession may not be used on any CP/DOS calls, e.g., DosSetPrty, which require a parent process/child process relationship.

Once a process has issued a DosStartSession specifying **Related**=1, no other process within that session may issue related DosStartSessions until all the dependent sessions have terminated.

Debugger Considerations:

Debuggers may want to debug all processes associated with an application no matter how the process was started, DOSExecPgm or DOSStartSession. A special trace option, TraceOpt=2, has been provided for this purpose. When a TraceOpt of 2 is specified the debugger must also supply the name of an existing queue and Related equal 1 on the DOSStartSession call.

The Session Manager will notify the debugger whenever a new session is created through DOSStartSession from the initial session started with TraceOpt equal to 2 or from any descending session. The queue is posted regardless of how the new session is started; related, independent, with or without inheritance. Sessions started without inheritance will be exec'd for tracing. It will be the responsibility of the debugger to resume the new processes execution through

PTRACE.

The debugger must issue `DosReadQueue` to receive notification when a child session is created. The word containing the request parameter, returned by `DosReadQueue`, will be one. The data element structure will have the following format:

SIZE	DESCRIPTION
-----	-----
WORD	Session ID
WORD	Process ID

`DosReadQueue`, with the **NoWait** parameter set to zero, should be issued by the debugger and this will be the only process which will have addressability to the notification data element. After reading and processing the data element, the debugger must free the segment containing the data element using `DosFreeSeg`.

The debugger may use `DOSSelectSession` to switch itself or any descendant session into the foreground whenever the current foreground session is a descendant of the debugger.

PgmName/PgmInputs Considerations:

The program identified by **PgmName** is exec'ed directly with no intermediate secondary command (CMD.EXE) process. Alternatively, the program can be exec'ed indirectly through a secondary command (CMD.EXE) process by specifying CMD.EXE for **PgmName** and by specifying either /C or /K followed by the drive, path, and filename of the application to be loaded for **PgmInputs**. If the /C parameter is inserted at the beginning of the **PgmInputs** string, when the application program terminates, the session will terminate. If the /K parameter is inserted at the beginning of the **PgmInputs** string, when the application terminates, the operator will see the CP/DOS command line prompt displayed. The operator can then either enter the name of another program or command to execute or enter the CP/DOS Exit command to terminate the session.

When the **PgmName** address is zero or the ASCIIZ string is null, the program identified by the **PgmHandle** will be started in the new session. If the **PgmHandle** is not specified the program specified as a parameter to the Shell on the PROTSHELL statement in the configuration file, CONFIG.SYS, is exec'ed and passed the specified **PgmInputs**. The default is the program name for the command processor, CMD.EXE.

The **PgmName** and **PgmInputs** strings combined length may not exceed 1024 characters.

Program Handle Considerations:

If a process issues a `DosStartSession` specifying only the program handle, then it must change to the working directory **before** it issues the `DosStartSession`, and start the new process inherited. If a process is started non-inherited, it will be up to that process to change to the correct directory.

Parent/Child Termination Considerations:

The parent must create the termination queue prior to specifying the queue name on a `DosStartSession` call. The Session Manager will continue to notify the parent session through the specified queue as long as the process issuing the `DosStartSession` call remains a parent session. In other words, when all the child sessions for a particular parent session terminate, the termination queue is closed by the Session Manager. An existing queue name must be specified on the next `DosStartSession` call, if the caller wants to continue receiving termination notification messages.

The CP/DOS session manager will write a data element into the queue specified when any child session terminates. The queue is posted regardless of who terminates the child session (for example, child, parent, or operator) and whether the termination is normal or abnormal.

A parent session issues `DosReadQueue` to receive notification when a child session has terminated. The word containing the request parameter, returned by `DosReadQueue`, will be zero. The data element structure will have the following format:

SIZE	DESCRIPTION
-----	-----
WORD	Session ID of child
WORD	Result code

`DosReadQueue`, with the **NoWait** parameter set to zero, should be issued by the process which originally issued the `DosStartSession` request. This process is the only process which will have addressability to the notification data element. After reading and processing the data element, the caller must free the segment containing the data element using `DosFreeSeg`.

An application may use the termination queue for additional interprocess communication provided a unique request identifier is passed via the `DosWriteQueue` interface. Request identifier values 0 through 99 are reserved for CP/DOS. Request identifier values greater than or equal to 100 are available for application use.

When a child session terminates, the result code returned in the **TermQ** data element will be the result code of the program specified by **PgmName** assuming either

1. the program is exec'ed directly with no intermediate secondary command (CMD.EXE) process, or
2. the program is exec'ed indirectly through a secondary command (CMD.EXE) process and the /C parameter is specified.

Otherwise, the result code of CMD.EXE will be returned.

When a child session is running in the foreground at the time it terminates, the parent session becomes the foreground session. When a parent session terminates, any child sessions it created with **DosStartSession**, specifying **Related=1**, are terminated. When an independent session, created specifying **Related=0**, terminates in the foreground, the shell selects the next foreground session.

Grandchildren Considerations:

A session started through **DosStartSession** may in turn issue **DosStartSession**. The following rules apply:

- The **SessID** specified on **DosSelectSession**, **DosSetSession**, and **DosStopSession** may only be the **SessID** of an immediate child, not a grandchild, etc.
- If a bond is established between session A and its immediate child session B, and if another bond is established between session B and its immediate child session C, then if session A is selected, session C is brought to the foreground. Reference **DosSetSession** for a description of what establishing a bond means.
- When a session terminates, all of its descendants -- children, grandchildren, etc. -- are terminated.

PM Shell Interface

Document private API used by the PM Shell to bring up the DOS application Advanced Properties dialog.

Multitasking

This page is intentionally left blank.

Scheduler Architecture

This page is intentionally left blank.

Problem Description/Objectives

The Scheduler component of OS/2 v 2.0 has the following objectives:

1. Support existing 16-bit segmented OS/2 applications

All applications written for previous versions of OS/2 should run as well or better under OS/2 v 2.0.

2. Support flat model 32-bit applications

The full set of API for flat model 32-bit applications must be provided so that developers can write applications that will port easily to future versions of OS/2 perhaps running on non-80x86 processors.

Because of time constraints, most of the DLLs shipped with OS/2 v 2.0 will be substantially unchanged from OS/2 v 1.2. Therefore, it is necessary that 32-bit applications be able to interact seamlessly with these DLLs. In future versions of OS/2, while all or most of the DLLs may be 32-bit, they will need to support old 16-bit clients. Thus, it is necessary that an arbitrary application be able to use both 16- and 32-bit DLLs.

3. Work around 80386 errata and "design notes"

On common steppings of the 386, there are errata that must be taken into account by the system. There are also non-intuitive "features" of the chip of which the system must be aware. Those that affect the Scheduler component are described in detail later.

4. Remove or raise system limits on resources

System-imposed limits on resources in earlier versions of OS/2 have proved to be a stumbling block to developers. Examples are the limits on the number of threads per process (fifty-three) and on the number of threads in the system (512 in OS/2 v 1.2). These limits must be raised or removed altogether in OS/2 v 2.0.

5. Reduce the system's resident memory consumption

Earlier versions of OS/2 do not always run well on machines that do not have large amounts of RAM. A major culprit has been the system itself which consumes large quantities of resident (i.e., non-swappable) memory. OS/2 v 2.0 must reduce the system's resident memory requirements so that a user may get reasonable performance on a machine with two megabytes of RAM provided there is sufficient swap space. To that end, Scheduler code and especially data structures must be made swappable wherever possible.

6. Add new system features in a way that minimizes the impact upon reused system code.

Because of time constraints, it is necessary that OS/2 v 2.0 use significant amounts of code from earlier versions of OS/2, most notably the file system. Any changes made to the Scheduler component should be made in such a way that the impact on this existing code is minimal.

Solutions/Justification

In order to meet the objectives presented above, the Scheduler component for OS/2 v 2.0 will do the following:

1. Supporting existing 16-bit segmented OS/2 applications

The 16-bit interface presented by OS/2 v 2.0 must be functionally identical to that provided by OS/2 v 1.2 so that all existing applications run without modification. Because of the way that the 80386 behaves when making a ring transition to a 16-bit stack segment (see item "Support for the 80386 Ring Transition problems"), the high word of ESP will be zero upon return from the kernel. The return code from kernel APIs will be placed in EAX, not just AX, effectively zeroing the high word of this register. Neither of these changes affect an application that runs on the 80286. There is the remote possibility that they could adversely affect an application that requires the 80386 (i.e., an application with 16-bit code segments that uses overrides to use 32-bit instructions). This risk is considered minimal.

2. Supporting flat-model 32-bit applications

- a. Verification Layer

Earlier versions of OS/2 used the SCI mechanism to protect the kernel from invalid parameters passed in by applications. This method is not suitable for new APIs implemented in C, where parameters are passed on the stack and pointers have no privilege level associated with them. Therefore, new APIs will follow a standard set of conventions for passing parameters to the kernel. Code following these conventions will be referred to as the "Verification Layer".

b. 32 DLL Initialization routines

A mixture of 80286 and 80386 DLL's may be loaded or attached to a process as a result of a program or module load request, so the Scheduler component must be capable of executing a mixture of 16-bit and 32-bit DLL initialization routines.

c. 32 ExitList routines

A process may be attached to both 16- and 32-bit DLLs; these DLLs may register ExitList routines. The Scheduler component must be capable of executing a mixture of 16-bit and 32-bit ExitList routines.

d. Provide a general, efficient thread Sleep/Wakeup facility implementing both Multiple Wakeup (as in earlier versions) and Single Wakeup.

The new semaphore APIs for 32-bit applications require the ability to wake only the highest priority thread waiting for an event rather than all threads waiting for an event.

Making a linear search for an event i.d. is not sufficiently efficient in a system that can have more than 512 threads waiting on different events.

3. Working around 80386 errata and "design notes"

a. Support for 80386 Ring Transition Problems

These are problems relating to the updating of ESP on ring transitions.

b. Support for page faults within critical sections of protected mode code when interrupts are disabled.

Specifically for the PM application, support is required for page faults within intra-process critical sections. This prevents simultaneous access to PM internal objects by threads of the same process. Note that in PM there is no problem with simultaneous access to objects from threads of different processes since a thread's process must be the owner of the object (a PM concept) before the thread can access the object.

- c. Support for page faults during user stack switches

Page faults may occur when the user stack pointer is in an incomplete state because it is being changed (i.e. when SS has been updated but before ESP has).

4. Removing or raising system limits on resources

- a. Support for larger kernel stacks

Larger ring 0 stacks are required for several reasons. First, File System Drivers (FSD's) may be written in a high level language and will require more ring 0 stack. Second, page faults will be handled in the context of the requesting thread instead of a dedicated thread. The attendant I/O operations will increase stack consumption. Design must support larger sizes of ring 0 stack with minimal impact to existing kernel code.

- b. Expandable per-thread data

It must be possible to increase the amount of data maintained per-thread easily and without imposing new restrictions on the user's ability to create threads.

- c. No per-process limit on thread creation

The OS/2 v 1.2 limit of fifty-three threads per process is not sufficient for the projected type of application that will use OS/2 v 2.0. The number of threads a process may have should be restricted only by the number of threads the system can support.

d. Allow up to 4096 threads in the system

The ceiling on the number of threads the system will support shall be raised from 512 in OS/2 v 1.2 to 4096 in OS/2 v 2.0. As before, there is no explicit limit on the number of processes, but each process requires at least one thread.

5. Reducing resident memory consumption

Resident memory consumption will be reduced in the following ways:

- a. Some Scheduler code will be made swappable, including

- Process creation
- Thread creation
- Process termination
- Thread termination
- Exit List processing
- DLL initialization processing (LibInit)

- b. The TCB as known in older versions of OS/2 will be split into a small resident portion to be known as the TCB, and a larger swappable portion known as the Thread Swappable Data block (TSD). The TSD will include the thread's ring 0 stack which is the single largest item of per-thread data.
- c. The PTDA will be split into resident and swappable portions.
- d. Data used for DLL initialization will be made swappable.
- e. Data used for Exit List processing will be made swappable.

Design Overview

In this section we identify the design issues that arise in providing the solutions for meeting the requirements of the Scheduler component. We discuss the design of each solution in detail, and consider alternative designs or variations where appropriate, showing why the preferred design is being adopted in each case.

1. Supporting existing 16-bit segmented OS/2 applications
 - 16-bit API calls with an odd number of parameter words

To support kernel entry (via call-gate, interrupt or exception) from a point within 16-bit user code, 16-bit gates, as in OS/2 v 1.2, would suffice. However, 32-bit code requires 32-bit gates, since EIP and ESP could be larger than would fit in 16 bits. We could mix 16- and 32-bit call gates for the 16- and 32-bit APIs, but we are required to have 32-bit trap and interrupt gates, since an interrupt can occur at arbitrary points in user space (i.e., while executing either 16- or 32-bit code). It is desirable to use 32-bit call gates even for 16-bit APIs so that a single format of kernel stack frame is used throughout the kernel. (Note that ring 2 call gates continue to be 16 bits.)

Given that we use 32-bit call gates to enter the ring 0 kernel code, we have to consider the problem that arises when calling a 16-bit API with an odd number of parameter words. Because the parameter count contained in a 32-bit call gate specifies the number of dwords to copy to the more privileged stack from the less privileged stack, there is no way to copy the exact number of words for such an API. We must copy either one word too few or one word too many. There are several possible solutions to the problem.

- Use a thunk to make the number of parameter words even.

Kernel APIs with an odd number of words of parameters would be routed to a thunk in user space that would add an extra word to the base of the parameter frame and then call through a gate into the kernel, copying the caller's return address along with the dummy word and the parameters. A special SCI thop would replace the return address to the thunk with the return address to the caller. The figure shows what the thunk and thop would look like.

```
DosOddAPI proc far
    movzx    esp,sp           ; ESP.hiword = 0
    pop      ax               ; (AX) = return IP
    push     word ptr [esp]   ; Copy return CS down
    push     ax               ; Push return IP
    call     TOddAPI          ; Call into kernel
DosOddAPI endp

threadop fixaddr
    movzx    eax,[bp].kstk_args ; (EAX) = caller's IP
    mov      [bp].kstk_ssf.ssf_eip,eax ; replace thunk IP
    mov      ax,[bp].kstk_args+2 ; (AX) = caller's CS
    mov      [bp].kstk_ssf.ssf_cs,ax ; replace thunk CS
    inc      bx               ; advance thop pointer
    inc      bx
    jmp      cs:[bx]          ; execute next thop
endop    fixaddr
```

Sample thunk and SCI thop for "odd" 16-bit APIs

This method, as can be seen from the examples, consumes an additional ten bytes of the user's stack and adds overhead to the API.

- Copy one word too few and get the missing word from within the kernel.

The missing word on the kernel stack would always be the first parameter word pushed by the caller. There are two approaches to obtaining this word:

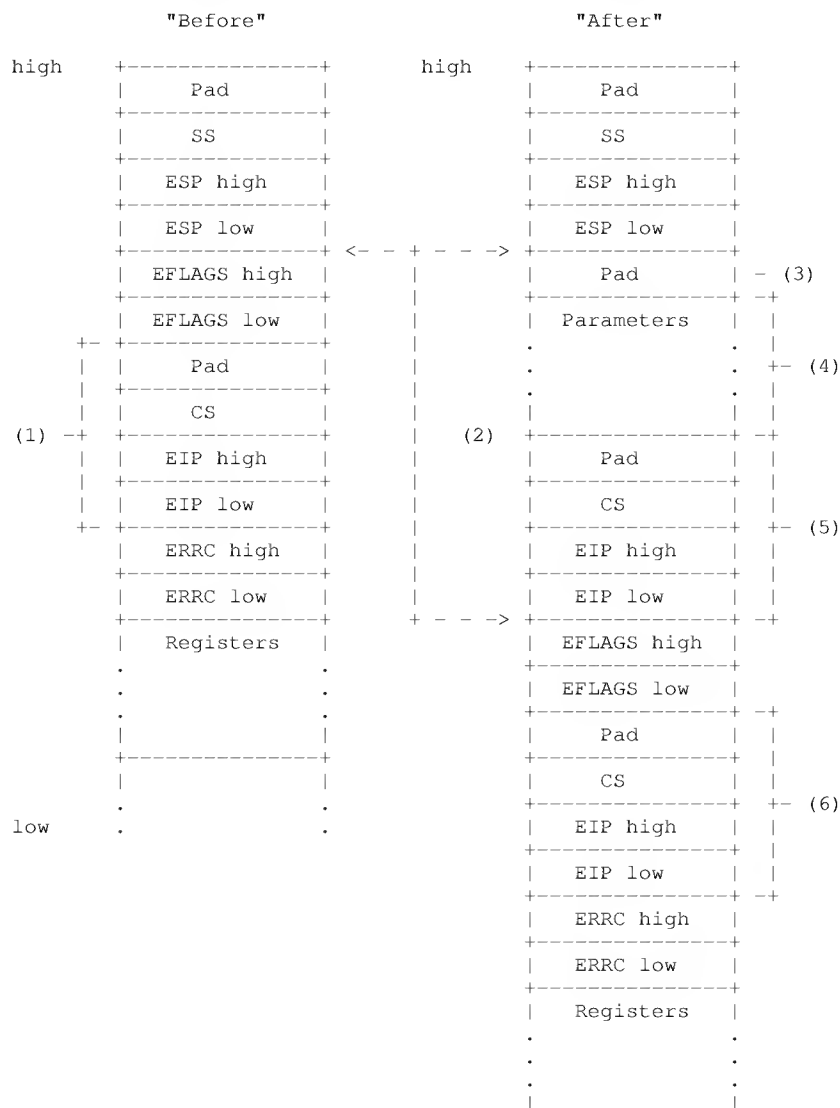
- a. Provide special thops to load the word from user memory into the appropriate register for the API worker routine. Since there are six general registers and two segment registers in which parameters are passed to workers, up to sixteen additional thops (eight for loading, eight for storing) would be added to the system. Further, each affected API would have to be inspected to see which thop needs to be invoked. The user's SP on the kernel stack would have to be incremented by two to account for the fact that the return to user space would otherwise pop one word too few from the user's stack.
- b. Fetch the missing word and fix up the kernel stack before executing the thop stream. There would be no need to write new thops or to change the existing thop streams, but the work involved in reformatting the kernel stack would be significant. All the parameters copied to the kernel stack along with the caller's return address would have to be copied down one word to make room for the insertion of the missing word. The return to user space would have to pop two bytes more than were actually copied to the stack by the call gate in order to account for the added word.

With either of these methods, the act of fetching the missing word could cause a page fault. Neither requires extra space on the caller's stack, but both add significant cost versus the same APIs in OS/2 v 1.2.

- Copy one word too many from the caller's stack.

The number of dwords copied by the gate is set so that one extra word is copied from the user's stack to the kernel stack. The user's SP on the kernel stack is decremented by two to account for the fact that the return to user space otherwise pops one word too many from the user's stack; this is the only additional overhead incurred by the caller.

This method works correctly except in one situation. If the caller's stack is "empty" apart from the parameters pushed for the API call, a stack fault will occur on the call, because it tries to fetch one parameter word beyond the end of the caller's stack segment. This scenario is extremely unlikely, but in the event that it occurs, there is a solution. The system emulates the faulting call instruction.





- (1) Address of instruction that caused stack fault.
- (2) Hole created for information from ring 3 stack.
- (3) Pad word needed to make even number of dwords of parameters.
- (4) Parameters copied from ring 3 stack.
- (5) Address in (1) adjusted to point AFTER faulting instruction.
- (6) Address of OS/2 call extracted from call gate.

Kernel Stack Before and After Call with Stack Fault.

When a stack fault occurs, an IRET frame including an error code is placed on the kernel stack automatically. The trap handler pushes all the registers; thus, yielding the "Before" picture in The figure.

Next, the trap handler calls a procedure that determines if the faulting instruction is a far (16:16) call. If it is, then a pointer to the location containing the target address is returned. If the target is a call gate, then it must be determined if the fault occurred because of a reason other than user error. If the following conditions are all satisfied, then the call was, in fact, a valid call and we must emulate it.

- a. The called function actually requires an odd number of parameter words.
- b. The proper number of words were actually pushed on the stack.
- c. The stack fault occurred because the need to round the word count up for the 32-bit call gate caused an attempt to fetch one word beyond the limit of the stack segment.

In order to emulate the call, we massage the stack to look like the "After" picture. In order to do this, we make room on the stack for the parameters, an extra word of padding, and a far return frame by copying everything on the stack below the user stack link down by the required amount of space. We move the return address from the IRET frame up just above the IRET frame and increment the EIP value by the length of the call instruction. We change the address in the IRET frame to be the address of the DOS function obtained from the call gate descriptor. Next, we use the user stack link to copy the parameters onto the kernel stack above the return address.

At this point, we are ready to dispatch to the OS/2 function. We pop all the registers to restore the values they had when the trap occurred, we add 4 to SP to remove the error code from the ring 0 stack, and we then do an IRETD. We will now start executing the OS/2 function, and the stack will look as though the call proceeded normally. Because of its very low overhead in all but the rarest of cases, OS/2 v 2.0 will make the gate copy one word too many and emulate the call in the case of spurious stack faults.

2. Supporting flat model 32-bit applications

- a. Interfacing to 16-bit API routines from Flat Model code.

Flat model user code may use EIP and ESP values greater than or equal to 64K. However, directly calling the 16-bit API when EIP or ESP values are greater than or equal to 64K cannot be supported for two reasons. First, the callee may be entered via a 16-bit far call, in which case CS:IP, not CS:EIP, is pushed on the stack for eventual return. Second, the callee may call a ring 2 routine through a 16-bit call gate. In this case, SS:SP and CS:IP, not SS:ESP and CS:EIP, are pushed on the ring 2 stack for eventual return to ring 3.

Indirect calling of the 16-bit API from flat model user code is supported via a set of 0:32 interface routines. Each interface routine translates 32-bit Flat addresses and 32-bit parameters into segmented 16:16 addresses and 16-bit parameters, and calls the appropriate 16-bit API routine. Refer to "32 Bit DOSCALLS Design Workbook" for the details of this mechanism.

- b. Access to Data Segments Larger than 64K Bytes from 16-bit API.

Access to a 64K byte "window" within a data segment larger than 64K bytes on a 16-bit API call can be provided by using an alias data selector that references the start of the window. A mechanism will be provided for the user to obtain an alias data selector to a given address in a data segment.

- c. Support for parameter validation

It is an OS/2 design requirement that the kernel not crash as a result of an incorrect operation by an application, such as passing in an invalid pointer. In OS/2 v 1.x the responsibility for ensuring that this didn't happen was with the SCI, or System Call Interpreter, which would only access user memory via a controlled mechanism. Objects were copied to kernel space from user space on entry to the API call, and copied back to user space on exit. The SCI thereby insulated the body of the kernel from the problems of having to continually check the validity of a user pointer. An internal subroutine (called a "worker") could freely dereference any pointers passed to it, because there was no question as to their validity. This is a desirable feature that we wish to preserve in OS/2 v 2.0; SCI will continue to be used for 16-bit APIs. New 32-bit APIs will be written in C, they will receive their parameters on the stack, and, for the most part, they will deal with 0:32 pointers. While it would be possible to write an SCI-like interpreter to perform parameter validation for such APIs, we choose not to do so for performance reasons. Therefore, there will be a piece of code between each 32-bit API's call gate and its internal worker that will perform the validation services previously provided by the SCI. For each 32-bit API, call this piece of code the 'validation layer'.

Objectives:

The validation layer is responsible for most accesses to and from user memory. We can divide the pointers to objects typically passed by apps to the system on the basis of how we would manipulate them in the kernel. The possibilities are to a) copy the object onto the kernel stack, b) copy the object into a kernel-allocated buffer, or c) manipulate the object in place. For simplicity we can refer to these as small, medium, and large sized objects.

Small and medium sized objects may be copied to the kernel stack or kernel memory relatively quickly. Since they are of known size, we do not have to worry about exhausting the size of the kernel stack. Large objects present more of a problem, because we do not want to suffer the performance hit of copying large amounts of data as part of the system call. Fortunately, these cases are rare and may be handled on a case by case basis within the worker.

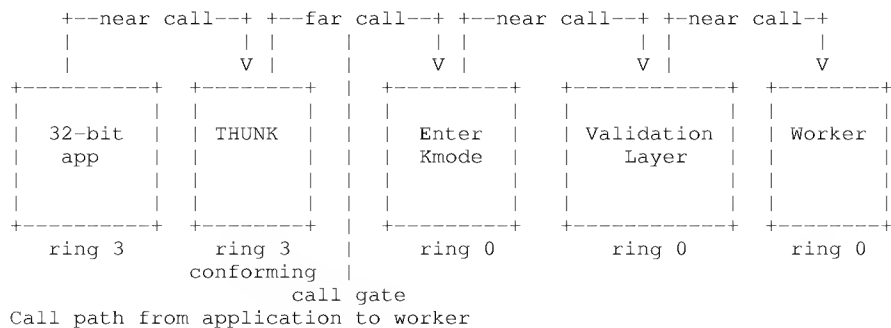
In summary, the validation layer provides the following services:

- Copying small and medium parameters to kernel space
- Calling the worker routine
- Storing any return values from the worker into user space
- Performing any error code translation that may be required

Entering and exiting kernel mode presents a problem, because the Enter/ExitKmode routines have to construct specific stack frames. There are two options: the validation layer may call Enter/ExitKmode directly, or it may be called by yet another routine that performs the transition to kernel mode. Having the validation layer call EnterKmode has the advantage that its parameters can be exactly those that are documented for the API call, but it makes it difficult for EnterKmode to ensure a consistent stack frame, and complicates plans for debugging support. Having the validation layer called by another wrapper routine requires that an extra parameter appear in the declaration of the validation routine to account for things pushed on the stack by EnterKmode.

As a side issue, if the validation layer does not call EnterKmode, it may be called directly from Ring 0 dynlinks when such things exist in the system. For this reason and those above, we have decided to go with the approach that EnterKmode will already have been called by the time the validation layer is executed. Note that this is no more or less expensive than having the validation routine call out to EnterKmode.

The complete path from application to worker is as diagrammed below.



Recall that the Far Thunk is required to effect the transition from ring 3 to ring 0.

Naming conventions:

There are four procedure calls between application and worker. In order to simplify and document the various paths into the kernel, we must adopt a naming convention to help group together related interface routines.

The application will be making calls to published API names. This means that a call to the published name (ie DosCWait) must go to the far thunk code. The far thunk code will then have to make a far call to some name that is associated with a call gate to enter the kernel. This code will then call EnterKmode and the validation routine, which will in turn call the worker. We will adopt the following convention:

DOSNAME will be the far thunk entry point.
TNAME will be the name exported from the kernel, called by the far thunk.
VRName will be the name of the verification routine.
(component abbreviation)ApiName will be the name of the actual worker (eg TKApiCwait)

Naming Convention for 32-bit APIs

The loader will be required to verify that callers of the T* names are from the DOSCALLS library.

In each validation routine there may be local variables that correspond to parameters. We have adopted the simple convention that the local variable should have the same name as the declared name of its corresponding parameter, with capital 'L' appended. This allows easy visual identification of which local variable is replacing which parameter.

```

int VRCWait(
    THUNKSEF      kframe,          /* placed on by KMEnterKmodeCallGate */
    USHORT        ActionCode,      /* 0 = Wait for task AND it's children */
                                /* 1 = Wait for task OR it's children */
    USHORT        WaitOption,      /* 0 = Wait for termination */
                                /* 1 = Don't wait for termination */
    PRESULTCODES  pRetCodes,       /* @pointer to place result codes */
    PPID          pRetProcessID,   /* @pointer to place PID of result code */
    PID           ProcessID        /* PID of desired task */
)
{
    RESULTCODES RetCodesL;          /* output */
    PID          RetProcessIDL;      /* output */
    int          rc=0;

    /* call CWait worker, passing in kernel addresses */
    if (rc = TKApiCWait(ActionCode, WaitOption, &RetCodesL, &RetProcessIDL,
                        ProcessID))
        goto error;

    /* if CWait succeeded, copy result codes structure to user space */
    if (rc = TKSuBuff(RetCodesL, pRetCodes, sizeof RESULTCODES, TK_UM_FATAL))
        goto error;

    /* if that succeeded, copy PID to user space */
    rc = TKSuWord(&RetProcessIDL, pRetProcessID);

error:
    return rc;
}
Sample validation layer code for DosCWait

```

```

USHORT VRCreateSem(
    THUNKSEF      kframe,          /* placed on by EnterKmode */
    USHORT        NoExclusive,     /* 0 = Exclusive ownership */
                                /* 1 = No exclusive ownership */
    PHSYSSEM      SemHandle,       /* @pointer to place semaphore handle */
    PSZ           SemName          /* @pointer to ASCIIZ semaphore name */
);
{
    HSYSSEM SemHandleL;            /* output */
    PSZ      SemNameL;             /* input */

    /* copy input string into kernel space */
    if (rc = getusrpathname(SemName, SemNameL))
        goto error;

    /* make the call */
    if (rc = TKApiCreateSem(NoExclusive, &SemHandleL, SemNameL))
        goto error;

    MemFree(SemNameL);

    /* copy HSYSSEM struct to user space */
    rc = TKSuDword(&SemHandleL, SemHandle);

error:
    return rc;
}

int getusrpathname(src, dest)
char *src;
char *dest;
{
    int rc;

    /* Allocate memory for a copy of the string */
    if (rc = MemAlloc(dest, PMPATHLEN))
        goto error;

    /* attempt to copy the string into kernel space */
    rc = TKFuBuff(src, dest, PMPATHLEN, TK_UM_FATAL)

error:
    return rc;
}
Sample validation layer code DosCreateSem

```

Compatibility details

- Support for 16:16 and 0:32 DLL Initialization

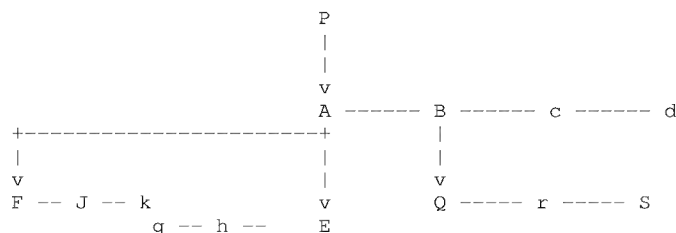
The address of a loaded Dynamic Link Library (DLL) module's initialization routine is contained in the Module Table Entry (MTE), created by the Program Loader when the module is first loaded. During a module load, the address of its DLL initialization routine (if any) will be passed to the Scheduler component along with its associated type through a Program Loader interface.

Since a mixture of 16 and 32-bit DLL's may be loaded, a mixture 16:16 and 0:32 DLL init routines may be executed following a module load or attach. The Program Loader interface is used to determine the type of the DLL init routine. Depending on the type of DLL init routine, the corresponding dispatcher will be setup.

The DLL Init data structure is a per-thread structure that contains the MTE handles for all modules loaded/attached to by the thread's task as a result of a DosLoadModule or, in the case of the initial thread of a program, the DosExecPgm for the task. The handles are used for two purposes; notification of newly loaded/attached modules to a debugging task, and obtaining addresses of initialization routines for newly loaded/attached modules.

The DLL Init data structure consists of a N-way tree of data blocks allocated via the Kernel Heap Allocator. Each block has a block header which contains the flat address of the next block at the current logical level, a set of flags, count of MTE handles in this block, the number of handles dispatched so far. Apart from these, there is information kept for the purpose of cleaning up after a DLL init routine has failed. This consists of pointers to the ancestor node, sibling node and descendant nodes for each of the descendant. The following figure highlights the use of various links within the tree.

A process P loads module A. In the following discussion all modules in upper case have init routines whereas all modules in lower case do not. A has references to B, c and d. A's init routine loads modules E and f. E has references to g and h. F has references to J and k. B's init routine load in Q which has references to r and S.



At level 1 we have modules A, B, c and d. There are no ancestors and no siblings. There is a descendant of B which is a level 2 structure consisting of Q, r and S. This node has no descendants and no siblings. The ancestor of this level is level 1. One descendant of A at level 2 consists of E, h and g. The other consists of modules F, J and k. The nodes at E and F have a common ancestor viz. level 1 block. Note that the pointers point to blocks and not modules. The first module at any block is the module being loaded. The others are static references to this module. They have no descendants of level 2 however. F has a sibling pointer to E.

Following the block header is an array of MTE handles for newly attached DLL modules. A set of blocks are linked together for each level of MTEs. A DosLoadModule from a initialization routine will create a new level and hence a new set of blocks. These will be unlinked after ALL the modules at ALL the levels have completed their initialisation or some module has failed its initialisation. If Debugging is enabled, the unlinking is delayed until the debugger is notified of all the attached modules.

If a module fails its init routine, then ALL modules at that level and direct descendants are freed up. The caller of DosLoadModule gets ERROR_INIT_ROUTINE_FAILED error code.

The basic change of design of the DLL initialization data area from that used for OS/2 on 80286 achieves the following goals:

- Removal of segmentation and migration to a more portable design.
- Elimination of memory reallocation.
- An n-way tree against a simple linear segmented list.

- Support for 16:16 and 0:32 Exit List Routines.

To support both 16:16 and 0:32 Exit List routines, a per-task linked list of data blocks allocated from the kernel heap will be used. Each Data Block will contain the address of a single Exit List routine, together with the execution order class (a number in the range 0 thru 511) and flags. Currently, the only flag is an indicator of whether the Exit List routine address is far 16 or near 32. The data blocks will form a singly linked list, the linkage being by flat address and the head of the list being obtainable from the PTDA. Exit list data blocks appear in the list in ascending numeric order (i.e. decreasing order of priority) of execution class. A data block for a new routine is added to the front of any existing blocks for the class or, if none, immediately in front of the first block for the next numerically higher class or, if none, at the end of the list. When Exit List processing for the task begins, routines are run in

order of appearance on the list, thus, the routines belonging to a given class are run in LIFO order.

The above design has some benefits over the implementation of Exit List in segmented versions of OS/2. First, memory reallocation of the Exit List has been eliminated. Second, data movement due to compaction of the Exit List has been eliminated. Third, the system limit on Exit List data area size of 64K bytes has been eliminated. Fourth, memory wastage due to unused portions of Exit Lists (which were allocated in increments of 64 bytes) has been eliminated. A tradeoff in increased memory consumption per Exit List routine stored has been made however.

- Providing a general, efficient Single Wakeup facility

In OS/2 v 1.2, at most 512 threads were allowed in the system at a time. When a thread wished to relinquish the CPU until some event occurred, it would place an event i.d. in a table indexed by its slot number in the scheduler tables. Threads waiting on the same event i.d. were also queued in the order that they waited. When an event occurred, all threads waiting for it would be made runnable in the order that they were queued. This mechanism where all threads were awakened when an event occurred is known as multiple wakeup. It is a good mechanism in situations where multiple wakeup is desired or where there is seldom more than one thread waiting for the event. Unfortunately, OS/2 has many events that do not meet these criteria. One example is the case of a semaphore used to protect a critical section. When the semaphore is released, the thread with highest priority waiting for it should gain ownership. With multiple wakeup, this goal was achieved in effect by making all the waiting threads race for the semaphore. The fastest (the thread with highest priority) won, and all the losers had to go back to the starting line and queue up for the next race. This was a source of significant overhead. Another drawback of this method was that event i.d.s were not required to be unique. Just because a thread was awakened did not mean that the event for which it was waiting had happened. The thread had to check to see if the event had occurred and go back and wait again if it had not.

Another drawback of OS/2 v 1.2's multiple wakeup mechanism was its implementation. When an event occurred, the table of event i.d.s had to be searched linearly to find waiting threads. While this algorithm was suitable for fewer than 512 threads, it is not suitable in a system that allows far more threads.

OS/2 v 2.0 provides both multiple and single wakeup services, and it does so in a way that is more efficient than in OS/2 v 1.2. The interfaces are called TKSleep and TKWakeup.

TKSleep is used to wait for an event. The calling thread specifies an event i.d. and whether the event is single wakeup or multiple wakeup. If the event is a single wakeup event, then the event i.d. specified by the caller must be unique. There is no enforcement of this rule! If two different single wakeup events were to share the same i.d., the results would be potentially disastrous, as one event might cause the other's waiting thread to be awakened at the wrong time while its own waiting thread would remain asleep. Since the single wakeup service is available only within the kernel itself, it should be possible to guarantee uniqueness by choosing i.d.s carefully.

TKWakeup is used to signal that an event has occurred. The caller specifies an event i.d. and whether the event is single wakeup or multiple wakeup. Since the type of event i.d. is retained for each sleeping thread, it is not possible to wake a thread waiting for a multiple wakeup event with a single wakeup notification and vice versa. TKWakeup does have the ability to wake at once all threads waiting for a single wakeup event; the 32-bit semaphore component makes use of this feature.

Event i.d.s are no longer stored in a table indexed by slot number. Instead, hashing is used. The event i.d. is hashed into a table of pointers to TCBs. Each TCB contains an event i.d., an event type, and a link to the next TCB in the hash chain. Threads waiting on the same event are guaranteed to be adjacent to each other in the hash chain. Within a group of threads waiting on the same event i.d., the TCBs are sorted by priority. Within a subgroup having the same priority, the TCBs are kept in FIFO order. While this implementation is superior to the linear table of OS/2 v 1.2, it still has its shortcomings:

- Hash chains are one-dimensional.

All threads waiting on any i.d. hashing to the same entry in the hash table end up on the same linked list. Thus, in order to find a thread waiting on some i.d., one may have to first skip over an indeterminate number of threads waiting on completely unrelated events. The practical effect of this drawback is typically small, since the number of entries in the hash table is one eighth the number of threads the system allows. This means that if the system had as many threads as allowed alive and each was waiting on a different event, then one would expect the hash chains to be no longer than eight elements each provided the hash function is good. Obviously, the worst-case performance could be much worse.

- Changes in priority are not reflected in the hash chains.

When a waiting thread's priority is changed, its position in its hash chain is not updated accordingly. This affects single wakeup events only. It is possible that a thread could wait for an event when its priority is low, and while it is waiting, its priority could be raised to be higher than any of the other threads waiting for the event, yet those other threads would all run first. Fixing this problem by moving a thread in the list when its priority changes gives rise to another problem. No explicit time ordering is maintained. If a thread were moved to a new priority group, we would not know its age relative to the other threads there. It might have been waiting longer, or it might not have been waiting as long. We simply would not know. In order to handle this case correctly, we would have to maintain some sort of time stamp along with the event i.d. This could be as simple as assigning each thread an ordinal when it begins to wait. The first thread would get 1, the second 2, and so forth. The problem with this is that since threads waiting are not sorted by time, we would have to scan all the threads waiting on a particular i.d. in order to find the next ordinal to give out (note that a two-dimensional hash chain would give us a way around this problem). There is also the problem of handling counter wraparound (extremely rare).

- Working around 80386 errata and "design notes"

1. Support for 80386 Ring Transition Problems

The 80386 CPU presents special problems related to ring transition. The use of 16-bit call gates or 16:16 stack pointers in conjunction with 32-bit code (that uses the ESP register) gives rise to the following two problems, because on ring transitions the 80386 CPU does not always update the full ESP register (and in some cases corrupts bits 16-31 of that register).

- Problem 1 - On a ring transition (call or return) to a stack whose B-bit is clear, only bits 0-15 of ESP are updated. This is true regardless of the type of transition (call or interrupt through 16 or 32-bit gate, 16 or 32-bit inter-level return).
- Problem 2 - On a transition to an outer ring whose stack has the B-bit set, all 32-bits of ESP will be loaded from memory regardless of the operand size of the return instruction executed. Thus a 16-bit return (RETF or IRET) to a stack whose B-bit is set will load the high 16 bits of ESP with garbage since the return stack link contains only the low 16 bits of ESP.

If all ring transitions went through 32-bit gates, all stack pointers were 32-bit (i.e. the stack selector descriptors had the B-bit set), and all code was 32-bit code using ESP, not SP, there would be no problem. However, this is not the case, and we therefore require some restrictions on ring 0, ring 2 and ring 3 stacks.

- Kernel stack descriptors will always have the B-bit clear.

The B-bit must be clear in order to use expand-down segments effectively to catch kernel stack overruns. An expand-down segment can have an upper limit of either 64K - 1 if the B-bit is clear, or 4G - 1 if the B-bit is set. The lower limit is controlled by the G-bit in the expand-down descriptor. If G is clear, the lowest addressable offset is the value of the 20-bit limit field plus one. If G is set, the lowest addressable offset is the value of the limit field plus one, multiplied by the page size. Thus, if G is clear, it is possible to have a byte-granular lower limit but only if that lower limit is 1M or less. Leaving G clear in an SS descriptor with B set makes virtually all of the machine's virtual address space accessible through SS (all SS-relative offsets from 1M to 4G - 1 are valid as far as the segmentation hardware is concerned). Protection from random pointer dereferences is virtually nil. On the other hand, setting G when B is set allows us to limit addressability through SS, but only if we page-align the stack and address it with offsets larger than 64K. The latter of these two restrictions is unacceptable, as it would force us to change all uses of BP as a frame pointer in existing 16-bit kernel code to EBP instead.

Note that having the B-bit clear for the ring 0 stack does not restrict kernel code from using ESP as top-of-stack pointer. This will in fact gradually become more common as the kernel is being converted to use 32-bit near model C code. Provided we ensure that the upper word of ESP is explicitly cleared on all kernel entries before it is used, no problem will occur with ESP references. Clearing the upper word of ESP can be done by inserting MOVZX ESP,SP instructions at strategic points. Specifically, we can insert MOVZX ESP,SP instructions into the kernel entry points as indicated.

- Insert one MOVZX ESP,SP only to catch all calls through gates that use SCI

A single MOVZX ESP,SP placed at the beginning of the SCI procedure would catch all such calls.

- Insert one MOVZX ESP,SP for each gate that bypasses SCI

This would include all 32-bit APIs as well as all 16-bit APIs that bypass SCI for performance reasons. The former are handled by placing a MOVZX ESP,SP in the EnterKmode macro used by all 32-bit APIs, and the latter have to be handled individually, but they are few in number.

- Insert one MOVZX ESP,SP to catch all interrupts

A single MOVZX ESP,SP placed at the beginning of the common interrupt handler routine would catch all interrupts.

- Insert one MOVZX ESP,SP for each exception type.

One MOVZX ESP,SP instruction would be placed at the beginning of each routine whose address appears in the IDT as an exception handler.

- Descriptors for Ring 2 stacks will always have the B-bit clear.

Such stacks are intended solely for the use of 16-bit code. The only time 32-bit code will be executed at ring 2 is when a call is made from a 16-bit ring 2 code segment to a 16-bit API that may legally be called from ring 2 and whose underlying implementation is 32-bit. In such a case, there will be a 16- to 32-bit thunk layer that will switch from the 16-bit ring 2 stack with B-bit clear to the flat stack via a DPL 2 descriptor with the B-bit set.

- Ring 3 code calling 16-bit API or making a 16-bit call into ring 2 must use a 16-bit stack pointer with the B-bit clear.

Ring 3 code that makes a 16-bit call into ring 2 code will not function correctly after return from ring 2 back to ring 3 if the B-bit is set for the ring 3 stack descriptor. If the ring 3 stack pointer B-bit was set, a 16-bit

RETF from ring 2 to ring 3 would actually trash bits 16-31 of ESP (as noted in problem 2 above). The ring 3 code executed after the RETF would not execute implicit top-of-stack accesses (e.g. PUSH, POP) correctly since these instructions would use ESP because the ring 3 stack descriptor B-bit was set. Since the caller of a 16-bit API is not aware of the implementation of that API, and since some 16-bit API have ring 2 portions of code entered via 16-bit call from ring 3, we must make the present restriction apply to all 16-bit API calls.

Requiring that the ring 3 stack B-bit be clear for 16-bit API calls from ring 3 code therefore enables existing 16-bit API support routines (that may contain both ring 3 and ring 2 portions) to be used in OS/2 v 2.0 without the need to change them.

2. Handling page faults in PM when interrupts are disabled

The proposed method of handling this is as follows. If a page fault occurs while interrupts are disabled, prevent other threads of the same process from running while handling the fault.

If a page fault occurs in user code while interrupts are disabled, the page fault handler will recognize that interrupts were disabled (by examining the saved copy of the user's EFLAGS on the thread's ring 0 stack) and will call an internal routine to prevent other threads of the current task from running. Signals must be held. Immediately before returning control to the faulting thread in user space, the fault handler will call another internal routine to re-enable intra-task thread switching and signal handling for the current task.

This design is of specific benefit to PM running in a paging environment in protecting critical sections. To protect critical sections in a non-paged environment, PM uses a combination of two techniques; disabling interrupts and "test and set" instruction sequences. Disabling interrupts allows a piece of user code (with IOPL) to execute to completion provided that no faults occur.

In non-paged versions of OS/2 we note that segment not-present faults occur only on segment register load instructions. If a segment currently referenced by a user selector is present and then becomes swapped out or discarded as a result of a kernel entry, that segment is forced to be present again immediately prior to returning from the kernel since the user's selectors are reloaded into the segment registers by the kernel code. It follows that in non-paged versions of OS/2 we can guarantee that a critical section of user code will execute to completion without interruption simply by doing the following.

- Do segment register loads before entering the critical section
- Disable interrupts before entering the critical section
- Do not make API calls, memory requests etc. that may cause the process to yield during the critical section

In a paged environment, however, page faults may occur in user code on any memory access. Further, interrupts must be re-enabled if a page fault is to be handled, since to keep interrupts disabled for the duration of the relevant I/O operations would lead to unacceptable dispatch latency.

3. Handling page faults during user stack switches

The only truly atomic way to switch stacks from one segment to another on the 80386 is using the LSS instruction. Unfortunately, this instruction is not present on the 80286, on which earlier versions of OS/2 run. All other methods require at least a two-instruction sequence to update both SS and SP. With the exception of LSS, the 80386, as with the 80286, will not acknowledge interrupts during the instruction immediately following an instruction that loads SS. This instruction should load SP. A correctly implemented 16-bit OS/2 application will use such a sequence to switch stacks. An application that does not adhere to this rule is broken and is not guaranteed to execute successfully in all situations. The problem on the 80386 is that a page fault may occur on the instruction that reloads SP, thus leaving the user's stack in an indeterminate state. This becomes a problem only when the kernel needs to set up a dispatch frame on the user's stack as the result of a signal or an exception during this window.

NOTE: There are some problems with the following discussion. Per-thread exception handling means that all threads are potentially affected, since termination exceptions are asynchronous. There is also the problem of how to handle a page-fault exception if it occurs while the stack is invalid (e.g., MOV SS,AX followed by MOV SP,[BX] where the page to which BX refers is invalid).

Further, not all instructions can be single-stepped at will. For instance, single-stepping a PUSHF means the user will have TF set in the flags image on the stack. This could cause a spurious (and fatal) trace interrupt later when the user does a POPF.

The problem of dealing with an entry into the kernel while the user stack is in an inconsistent state reduces to that of disabling signal handling for thread 1. For page faults, two methods were considered.

- Method A - Disable signal handling for duration of page fault

To disable signal handling, we set a flag, SIG_DISABLED, in a per-thread variable, from within the page-fault handler. The SIG_DISABLED flag is only needed by thread 1 of a process since signals are handled only by thread 1, but setting it unconditionally eliminates the need to check that we are in thread 1. The effect of the SIG_DISABLED flag, if set for thread 1 of a process, is to delay the handling of pending signals for which handlers are installed for the thread's process. Action can still be taken for pending signals that do not have a handler installed, since access to the user stack is not required in order to deal with these

signals.

The SIG_DISABLED flag would be cleared at the end of the page fault handler, and we would then return to execute the faulting instruction. A second page fault may occur, in which case we would again disable signal handling from within the page fault handler. This sequence of events would be repeated until finally no fault occurred and the faulting instruction was executed.

This scheme has the following problem. If an interrupt is acknowledged on return to ring 3 immediately prior to attempting to re-execute the faulting instruction, the SIG_DISABLED flag will be clear, so signal handler setup will occur when the interrupt handler leaves Kernel Mode after return to task-time processing following the servicing of the interrupt. The user stack pointer is still in an inconsistent state since the faulting instruction which loads eSP has not yet been executed. To overcome this problem, we would have to disable signal handling on entry to interrupt handlers, an unacceptable limitation. Signal handling would only become re-enabled on non-interrupt or non-exception entries into the kernel. This means that we would need to wait until the next call-gate before we handled a signal. This causes an unacceptable amount of latency for signal dispatch.

- Method B - Single-step the faulting instruction with signals disabled

As in Method A, we disable signal handling in the page fault handler if we take a page fault in thread 1. With handling disabled, we return from the page fault handler with single stepping enabled in the user flags (the TF flag is set on the transition to ring 3). If we immediately take another page fault or an interrupt, we are OK, since signal handling is still disabled. If we execute the faulting instruction, the user stack pointer is then in a consistent state since the instruction loads eSP. We then enter the trap 1 handler and unconditionally clear the SIG_DISABLED flag.

The main problem with this method is the additional overhead incurred on all page faults in thread 1 due to single stepping. There is also a second problem relating to breakpoints/dynamic tracepoints which we now discuss.

A problem occurs with this method if the user sets a breakpoint on the instruction that loads eSP. When a breakpoint/tracepoint is set, the first byte of the affected instruction is overwritten with an INT 3 byte (CCh). Normally, the INT 3 handler restores the replaced byte of the instruction, single-steps the instruction, and restores the INT 3 in the Trap 1 handler so that the breakpoint can be hit subsequently. If however a fault occurs when attempting to single-step the instruction, the fault handler restores the INT 3 and services the fault. On return, the INT 3 is re-executed and the cycle repeats until no more faults occur. Therefore, an attempt to single step the faulting instruction may result in single-stepping an INT 3 instruction instead. We must therefore keep signal handling disabled until we succeed in single-stepping the faulting instruction. There are two ways of dealing with this problem.

-- Document that breakpoints must not be set in middle of stack switch

We document that a user GP fault will occur if a breakpoint is set on the instruction following a load of SS.

-- Allow the breakpoint and handle the situation

A second per-thread flag, INT3_EXECUTED, could be used to indicate that the INT 3 handler was executed. The flag would be set in the INT 3 handler. For best performance, we would always set the INT3_EXECUTED flag whenever we enter the INT 3 handler even though we actually only need to set the flag if the thread is thread 1 and its process has signal handlers installed. Setting the flag unconditionally avoids the need to test for thread 1 and whether there are signals to be handled.

The INT 3 handler restores the original byte of the faulting instruction and leaves TF set so that the trap 1 handler can restore the INT 3 breakpoint. Modify the trap 1 handler so that it clears the SIG_DISABLED in all cases except where an INT 3 was single-stepped within thread 1 of a process.

```
IF NOT (THREAD1 and SIG_DISABLED and INT3_EXECUTED)
    clear SIG_DISABLED
ENDIF
Clear INT3_EXECUTED
Trap 1 Handler Logic to deal with INT 3 Breakpoints
```

- Method C - Selectively single-step the faulting instruction

Instead of unconditionally setting the SIG_DISABLED flag in the page fault handler if we are thread 1, we attempt to determine whether we really need to have signal handling disabled. To do this, we insert the following logic into the page fault handler

```
IF (thread 1)
    IF (fault was caused by a CS page)
        Set SIG_DISABLED and TF
    ELSE
```

```

        Read faulting instruction /* may cause fault */
        Parse faulting instruction
        IF (instruction modifies eSP)
            Set SIG_DISABLED and TF
        ENDIF
    ENDIF
ENDIF
Logic inserted into page fault handler to handle signal disabling.

```

We insert the same logic into the Trap 1 and Trap 3 handlers as indicated in Method B.

- Justification for the Preferred Method

Method B is the preferred method, with the additional logic for handling INT 3 breakpoints within the stack switch, for two reasons. First, Method B is reasonably simple to execute. By contrast, Method C needs an instruction parser that could be quite complicated. The overhead of the single stepping on thread 1 page faults is small compared to the overhead of parsing out a faulting instruction to see that it affects the eSP register. Second, Method B avoids the problems of Method A in which signal handling would have to be disabled during interrupts.

• Removing or raising system limits on resources

1. Removing the per-process limit on thread creation

The limit of about fifty threads per process in OS/2 on 80286 is an artifact of the organization of PTDA's and TCBs. The TCBs for all of the threads of a given process were contained in the same segment that contained the process' PTDA. Since a segment could be at most 64K bytes long, the number of threads a process could have was limited to the number of TCBs that would fit in 64K minus the size of the PTDA.

On the 386, two ways of removing this limit were considered:

- Allow the segment containing the PTDA and the TCBs to grow beyond 64K bytes. A segment on the 386 can be as large as 4G bytes, so the hardware does not prevent us from putting more than fifty TCBs in one segment. This method has three major drawbacks:
 - Existing 16-bit code in the kernel would have to be rewritten to use ESP instead of SP as the stack pointer and EBP instead of BP as the frame pointer. This would be a substantial amount of work.
 - The use of a stack segment larger than 64K bytes would require having the B-bit set in the kernel SS descriptor. This is unacceptable for reasons mentioned earlier (see item "Ring Transition problems").
- Separate the PTDA's from the TCBs. Don't put them in the same segment. This method works extremely well for new kernel code that is written in C and is 32-bit and views the machine's address space as a flat, 32-bit space. But two questions arise. How to support existing 16-bit code that relies on both the PTDA's and the TCBs being addressable through SS? How to provide stack overrun detection?

The solution is take advantage of the paging hardware to map the current PTDA and the current TCB into a contiguous 64K region of virtual memory and use the segmentation hardware to map this area with a B-bit-clear, expand-down SS selector whose limit is set to the base of the kernel stack in the TCB. Thus, existing 16-bit kernel code can address the current PTDA and the current TCB just as it always has. Some small problems arise, however:

- Existing 16-bit code that walks a process' chain of TCBs must be rewritten since these TCBs no longer reside within the same segment. There is only a small number of affected routines.
- New 32-bit code must be able to cope with the fact that its frame variables (parameters and locals) are not normally addressable via DS. Fortunately, the C compiler supports the generation of code that does not assume SS and DS are the same. There still is the problem of passing a pointer to a frame variable to a function that expects a pointer that is relative to DS, the flat address space. Fortunately, this, too, is easily solved. Note that a frame variable has two addresses: its address relative to SS and its address in the flat address space (its "true" address). For any frame variable, these two addresses differ by a value that is constant for that particular thread. This constant can be computed when the thread is created and stored in its TCB. When we context-switch to that thread, we extract the value from the TCB and store it in a global variable accessible by the C code. Thus, to convert the address of a stack variable into its true address, a C routine simply takes the address of the variable relative to SS (given by the '&' operator) and adds the value in the global variable to it. The result is a flat address that may be passed to any function expecting a flat address.

The second method is the method of choice for OS/2 v 2.0. If the PTDA's and TCBs are no longer in a segment together, how should they be organized? The approach chosen for OS/2 v 2.0 is to place them in 32-bit BMP objects. At initialization time, the maximum number of threads allowed in the system is determined from examining config.sys, and this value is stored in the variable MaxThreads. A BMP object large enough to hold MaxThreads PTDA's is allocated, and so is one large enough to hold MaxThreads TCBs. The use of BMP allows these structures to be

packed together to reduce memory waste, since neither the size of the PTDA nor the size of the TCB is an exact multiple of the page size.

Packing the PTDA and TCBs together so that a single structure may overlap a page boundary creates a minor, but significant, protection hole. The SS region maps not only the current PTDA and current TCB, but also the parts of PTDA and TCBs that share pages of physical memory with them. In OS/2 on 80286, an errant thread could trash any or all of its sibling threads' TCBs as well as its own PTDA. Under OS/2 v 2.0, an errant thread may trash parts of some possibly unrelated threads' TCBs and parts of some possibly unrelated processes' PTDA. Note that such a problem could only be caused by a bug in the kernel itself or in some piece of trusted code like a device driver or a FSD. This risk is considered acceptable given that the alternative is to incur the considerable waste associated with making the sizes of PTDA and TCBs page multiples.

2. Allowing up to 4096 threads in the system

Tables indexed by slot number must be made large enough.

- Reducing resident memory consumption

The goal of reducing resident memory consumption is achieved in the Scheduler component by doing the following things:

1. Making Scheduler code swappable, including

- Process creation
- Thread creation
- Process termination
- Thread termination
- Exit List processing
- DLL initialization processing (Liblnit)

2. Making per-thread data swappable

Under OS/2 on 80286, there was a Thread Control Block, or TCB, for each thread in the system. Each TCB included the kernel stack for the thread and resided in resident memory. Since a TCB was about 2K long and since there could be at most 512 threads, TCBs would consume at most 1 megabyte of resident memory. OS/2 v 2.0 allows up to 4K threads in the system at one time, and each kernel stack alone is 3K bytes long. If we were to organize TCBs as OS/2 on 80286 did, one would require over 12 megabytes of RAM just for TCBs if one actually wanted to have 4K living threads.

Requiring over 12 megabytes of RAM just for TCBs in order to support 4K threads renders the limit virtually meaningless, and, for all intents and purposes, imposes a much smaller practical limit on the number of threads the system can support. Consider that a large class of machines that will be running OS/2 v 2.0 cannot be configured with more than 16 megabytes of RAM today. Such a machine running 4K threads would have less than 4 megabytes of RAM available for the user's applications and for the system's code and other data structures.

Even in situations where the user is running far fewer threads, the memory consumed by TCBs is a significant expense. Most of the data in the TCB is not needed unless the thread is actually running, and at any given time under typical circumstances, most of the threads in the system are not even in the ready state. Only one thread is actually running at any given time. Thus, most of the RAM consumed by TCBs could be used for other purposes most of the time if only a TCB, or a portion of it, were swappable.

In OS/2 v 2.0, we split the TCB structure from OS/2 on 80286 into two separate structures. The first, called the TCB, contains all the per-thread data that cannot or should not be swapped out (e.g., Scheduler data referenced at interrupt time, fields that are referenced frequently by performance-critical code). The second, called the Thread Swappable Data block (TSD), contains all the other per-thread data, including the kernel stack.

- TSDs for protected mode application threads and VDM threads are allocated from swappable memory. TSDs for all other threads (internal threads like the Ager and monitor threads) are allocated from resident memory.
- No thread is allowed to be in the ready state unless its TSD is currently in memory.
- The Page Manager's Ager ages TSDs just as it ages other system memory with one exception. When the Ager selects a page that belongs to a TSD to be made not present, it first calls out to the Scheduler component.
- Upon being called by the Page Manager, the Scheduler component first checks to see if the corresponding thread is in the middle of swap I/O. If it is, then the Scheduler component returns an indication that the Page Manager should not idle the thread's TSD. Otherwise, it updates the state of the TSD in the TCB, and, if the thread is in the ready state, then the Scheduler component places it in a new unready state called the TSD state.
- All threads in the TSD state are kept in a priority queue called the TSD queue.
- A special internal thread called the TSD thread exists to service the TSD queue. This thread's job is to take all the threads in the TSD queue, make their TSDs present, and return the threads to the ready queue. The TSD thread runs at the same priority as the highest priority thread in its queue.

- If the priority of a thread in the TSD state is changed, then the position of that thread in the TSD queue changes as appropriate, and, if necessary, the priority of the TSD daemon is adjusted accordingly.
- When a thread makes the transition from an unready state to the ready state, the Scheduler checks the state of its TSD. If the TSD is not present, an attempt is made to reclaim it from the Page Manager's Idle List. If this fails, then the thread is put in the TSD state and placed on the TSD queue. If necessary, the TSD thread is awakened to process the threads in the queue.

There are two disadvantages to this scheme:

a. Loss of Starvation Information

When a process is moved from the ready state to the TSD state, it loses its starvation history. The actual effects of this loss are mitigated substantially by the way the starvation strategy is currently implemented. A thread in the normal priority class receives a starvation boost once every three seconds if necessary. This boost is granted for one time slice, regardless of whether the thread manages to execute any of the user's code. Even if we were to preserve the starvation history, or even to perform starvation accounting on the threads in the TSD queue, the TSD thread (since it takes on the priority of the highest priority thread in its queue) would benefit from that boost only until it blocks. This will happen almost immediately, since it has to hit the disk to swap in the TSD. At this point, the TSD thread will revert to normal priority, and it may not complete the swap-in operation until it receives its own starvation boost after three seconds. This scenario illuminates the need to reexamine the current starvation strategy.

b. Loss of FIFO Ordering Among Threads of the Same Priority

Since the Ager has no knowledge of the ordering of the threads in the ready queue, it may destroy the FIFO ordering of threads having the same priority. The first thread, within any priority level, moved to the TSD queue will be the first thread moved back to the ready queue by the TSD thread. Thus, if the last thread in line at a given level is moved to the TSD queue just before the first in line, it will be moved back to the ready queue just before the first one, too. It may end up actually running first. More investigation of this phenomenon in an actual system is necessary to determine if it is a real problem.

Two other swap-out policies were considered. The first is essentially the same as the policy described above with the following exception. When the TSD of a ready thread was made not present, the fact would have been noted by the Scheduler component, but that thread would not have immediately been removed from the ready state. Instead, when the Scheduler picked a new thread to run (GetNextRunner), the state of its TSD would have been checked at that point, and if it were unavailable and could not be reclaimed, then the thread would have been placed on the TSD queue and placed in the TSD state. Threads in the TSD state would have remained on the ready queue just as threads blocked by a critical section do today. The advantages of this policy were:

- a. threads that were otherwise ready but whose TSDs were not present would have continued to receive their normal starvation boost, and
- b. there would have been less work involved in the call out from the Page Manager to the Scheduler component.

The disadvantages were:

- a. there would have been extra code added to the GetNextRunner path, and
- b. there might have been additional unrunnable threads in the ready queue that GetNextRunner would have had to skip over.

Because of these disadvantages, this method was discarded.

The second policy considered was to lock down the TSDs of all ready threads. The advantages of this scheme were:

- a. no modifications to the Ager were required, and
- b. a thread could not have been forced out of the ready state by having its TSD made not present.

The disadvantages were:

- a. fewer pages would have been available to be aged,
- b. a working prototype of this method uncovered serious difficult deadlock problems inside the Page Manager, and
- c. the code for the prototype proved to be more complex than it was believed either of the other methods would be.

Because of these disadvantages, especially the deadlock problems, this method was discarded.

Within the method of choice, a couple of variants for TSD organization were considered. Since the TSD is just over 3K bytes long and a page is 4K bytes long, it was suggested that the TSDs be packed in a BMP object in order to save

physical storage. This method leads to some complications, however, all of which relate to the fact that TSDs might then overlap page boundaries:

- a. When the Ager claimed a page from the TSD region, it might render zero, one, or two threads unrunnable. The logic to figure this out would be somewhat more complicated, and the work involved in making the threads unrunnable could be as much as doubled.
- b. When the TSD thread went to make a TSD present, it might have to touch two pages, and in doing so, it might actually make as many as three threads runnable again. The logic to handle all the possible cases would be quite a bit more complicated. There is also the possibility (slight, but real), that, while bringing in the second page, the first would be swapped out again. This would occur only under extreme conditions, but there would have to be code in the system to deal with it.
- c. In the worst case scenario, with every other thread runnable and every other thread blocked, all the pages in the TSD region would have to be present because of overlapping; thus, more RAM would be consumed by TSDs than if they were page-aligned.

It was also suggested that the TSDs be allocated from the swappable heap. While this would most likely eliminate the worst case scenario in the item above, it has an additional problem of its own. Since the TSDs would not be concentrated in an array, but would be scattered throughout the swappable heap, there would be no simple way to determine whether a page wanted by the Ager actually contained part of a TSD, and if so, whose TSD it was.

For the reasons mentioned above, these variants were discarded.

3. The PTDA will be split into resident and swappable portions.
4. Data used for DLL initialization is allocated from the swappable heap.
5. Data used for Exit List processing is allocated from the swappable heap.

Data Structures Description

In this section we describe the proposed revised formats of the data structures that are managed by the Scheduler component.

- Per-Task Data Area (PTDA).

Each process has a Per-Task Data Area (PTDA) in which per-process data is stored. When running at task time at ring 0 inside the kernel, the SS selector maps the current thread's PTDA. The PTDA is mapped in at the very top of the SS region so that its fields may be accessed by fixed offsets relative to SS while at the same time allowing SS to be an expand-down selector in order to provide hardware stack overrun checking. The figure shows the layout of the PTDA.

PTDA - Per Task Data Area (PTDA) Definitions

PTDA Layout

- Thread Control Block (TCB).

A TCB exists for each thread in the system. The TCB contains per-thread information that must remain resident at all times. The figure shows the layout of the TCB.

TCB - Thread Control Block structure

TCB Layout

- Thread Swappable Data block (TSD).

A TSD exists for each thread in the system. The TSD contains per-thread information that may be swapped out when the thread is not running, including the thread's ring 0 stack. The figure shows the layout of the TSD.

TSD - Thread Swappable Data

TSD Layout

- Exit List Data Areas

A list of addresses of routines to be executed on termination of a process is kept in a system data area known as an Exit List. An Exit List is linked to the PTDA of the process that owns it. Prior to a process being marked to terminate, Exit List routines can be added or deleted from an Exit List via the DosExitList API call. The figure shows the format of a record in the Exit List.

EXITLIST - Exit list data structures

Exit List Record Layout

- TCBPtrs array

The TCBPtrs array is an array of MaxThreadsInSystem DWORDS indexed by thread number. For thread n, TCBPtrs[n] contains the linear address of the TCB for the process to which thread n belongs.

- TaskPtrs array

The TaskPtrs array is an array of MaxThreadsInSystem WORDS indexed by thread number. For thread n, TaskPtrs[n] contains the handle of the PTDA for the process to which thread n belongs.

NOTE: This array is retained in the short term to enable existing kernel code based on the 16:16 kernel memory model to be used. The intention is to phase out use of this array and instead use the PTDAPtrs array.

- DLL init data blocks

One of these is created whenever a thread does a DOSLOADMODULE or DOSEXECPGM call for which one or more modules in the module graph require an initialization routine to be run. DLL init data areas are allocated using the swappable Kernel Heap Allocator instead of allocating/reallocating the segments. These blocks will be chained with its head in the threads TCB and would hold MAXHDL (currently fixed at 16) MTE handles each. At the end of the LibInit, i.e. when the dispatcher has run through the init routines, these blocks are unlinked and freed. The figure shows the format of a LibInit record.

LibInit block structure

LibInit structure for module unload notification

LibInit Record Layout

Local Procedures

The following functions can be called from anywhere in the kernel, subject to the restrictions listed in the header for each function.

TKCreateVDMProcess - create a process for a new VDM

TKCreateVDMThread - create a thread for an existing VDM

TKCheckExitListRef - Check for References to Modules From ExitList

TKTermTask Terminate current task

TEMPORARY: We call ProcessTerminate to do the job

ENTRY: None

EXIT: Does not return

_TKTermThread - Terminate this thread. (after exception handling)

_TKTermThread is called from _TKExit, and from the Exception Manager as the default action of fatal exceptions. _TKTermThread terminates the current thread. It may only return when thread 1 attempts to run exitlist.

void _TKTermThread ();

ENTRY NONE

EXIT Void. Sometimes does not return.
_TKTermThread MAY RETURN for thread 1 of a process that has an exitlist routine, or as we run exitlist. No other threads return.

Change history:

02/16/94 Skip Nizinski 69005. Check if terminating thread is involved with any current hard error pop up message and set indicator in _SwapFullPTDA and _HardErrTCB globals.

TKForceTask - Set Force Flags for Task/Force Task to be Runnable

TKForceThread - Set Thread into Force Pending State

TKEachThread - Call a worker function for each TCB of a process.

TKTidTo_pTCB - From Thread ID, find TCB Pointer

TKTidTo_iSlot - From Thread ID, find Slot index

TKiSlotTo_pTCB - From Slot Index, find TCB Pointer

TKInnerExit - Exit this thread. (before exception handling)

TKDieSoonPTDA - Mark all threads in a process to die.

TKExit - Exit this thread (and maybe process)

TKForceCritSec - Handle the TK_FF_CRITSEC force flag

VRExit - API Worker for DOSEXIT - Exit thread

VRResumeThread - API entry for DOSRESUMETHREAD

VRSuspendThread - API entry for DOSSUSPENDTHREAD

TKLibiRecordMTE - Record applicable MTE handles in LibInit block

TKLibiStartDispatch - Setup initial dispatch to LibInit dispatcher

TKLibiError - Free the level of libinit records being built

TKLibiInitNextDLL - Select next LibInit routine for LibInit Dispatch

TKLibiCheckRef - Check if the MTE handle is recorded by any thread.

TKLibiSetupUnload - allocate LIBF block for this thread

TKLibiRecordUnload - Record a module being detached

TKLibiUnloadDone - Notify DosDebug of all the modules being detached

TKLibiDbgNextDLL - Pass the next MTE handle to Debug

TKLibiDbgDone - Report Debug notification completion to LibInit

TKPidToPTDA - Convert Process Id to PTDA address

TKScanTasks - Scan a group of Tasks using a kernel worker routine

TKLinkTask - Link new task's PTDA to sibling and parent PTDA's

TKUnlinkTask - Unlink PTDA from sibling and parent PTDA's

TKAdoptTask - Link process into adopter's PTDA chain.

TKDescendant - Check if a given process id is the process id of a descendant of the current process. **TKPidGetSem** - get PTDA semaphore given a Pid

TKWaitThread - Wait on a thread's death

TKCreateThread - create a new thread

TKAllocR2Stacks - allocate ring 2 stacks for all threads

VRCreateThread - entry point for DosCreateThread

VR32CreateThread - kernel entry for Dos32CreateThread

VR32KillThread - kernel entry for Dos32KillThread dcr 1420

VR32VerifyPidTid - kernel entry for Dos32VerifyPidTid

VR32WaitThread - Entry point for Dos32WaitThread API

TKUpdateLocalFPHandler - update 16-bit fault handler address

VR32QUERYTHREADCONTEXT - Examine context of a suspended thread

TKSuFuBuff - copy from user memory to user memory

TKSuBuffz - Copy null-terminated string to user space

TKFuBuff - Copy from User Memory to Resident System Memory

TKFuBuffz - Copy null-terminated string from user space

TKFuBufLen - Common code for TKFuBufLenz and TKFuBuffLenz

TKUshrMemPageFault - Handle Erroneous Page Fault in User Memory

TKUshrMemGPFault - Handle Erroneous GP Fault in User Memory

VR32QuerySysInfo - Query System information (API entry)

TKPreInit - Perform System Initialization for Tasking (1st stage)

TKSchedInit - Initialize Scheduling Data Structures

TKInit - Perform System Initialization for Tasking (2nd stage)

VRSleep - Worker for DosSleep

TKSleep - Block the current thread from running.

TKWakeup - release blocked threads

TKWakeThread - wake up a single (specific) thread

TKQueryWakeup - query which thread(s) would be awakened

TKIdleTSD - Make a TSD not present

TKNeedTSDDaemon

TKTomTimeslice - Handle a timeslice - expired TOM event

TKTomWake - Wake a thread when it times out.

TKTomStarve - Starvation TOM worker

TKMakePresentKDB - make a page or TSD present for the kernel debugger

Tasking Exported Procedures

The following functions can be called from anywhere in the kernel, subject to the restrictions listed in the header for each function.

tkSetPIB - set the process' PIB

tkAllocPIB - allocate the PIB for the process

tkAllocPTDA - allocate and initialize a PTDA

tkEvapPTDA - free the given PTDA

tkExitList - dispatcher for 16- and 32-bit exit list functions

tkAddExitListRoutine - Add Exit List Routine for Current Task

tkDeleteExitListRoutine(pRoutine, Class, Type)

tkNextExitListRoutine - Setup to Run Next Exit List Routine

tkCleanUpExitList - frees the exit-list for this process

VRExitList - API entry point for 16-bit exitlist routines

VR32ExitList - API entry point for 32-bit exitlist routines

tkAsyncPTERM - Post asynchronous PTERM to a thread

tkPostPTERMs - Post asynchronous PTERMs to every thread

tkLibiInitStatic - Initialilize TKLibiCur structure

tkLibiCleanup - Free all the libi data blocks

tkLibiFreeHeap - Free all the heap associated with this level down

tkLibiFreeModules - Free Modules if there are any waiting to be freed

tkLibiCheckReference - Check if module recorded at this level or lower

tkGetLibiDispatcherAddr - Initialize pointers to the dispatchers

tkStartScan - notification that TKScanTasks is starting

tkStopScan - notification that TKScanTasks is stopping

tkWaitScan - wait for TKScanTasks to complete

tkCheckPid - Worker Routine for TKPidToPTDA

tkCheckCsid - TKScanTasks worker to check if the given process

has the same Csid

TKGetReadyThread - Get the highest priority ready thread

TKIdleLoop - Perform Idle-Loop Processing

TKUpdatePriority - Update thread priority after a possible change.

TKUpdateState - Update thread state after a possible change.

TKYieldNonIdle - Yield to Non-Idle threads B733890

B733890

tkTSDDaemon - Make all TSDs in the TSD queue present

TKExitInversion - Exit Priority Inversion Protected Section

TKDeclareInversion - Check for a Priority Inversion

tkKDBDaemon - make a TSD present for the kernel debugger

tkVerifyTCBLinks - verify that TCB links are valid

tkEvapAll - clean up dead TCBs and dead PTDA's

tkLinkTCB - link TCB onto a task's TCB list

tkUnlinkTCBSub - unlink TCB from specified PTDA's TCB list

tkRunWaitThreads - Run all threads waiting for us to die

tkAllocTCB - Obtain a new Thread Control Block

tkFreeTCB - free a Thread Control Block

tkUnlinkTCB - unlink TCB, mark it for freeing, make thread unrunnable

tkSetPTDAVars - set up PTDA variables

tkFreeUserStacks - free the thread's user stacks if any

tkFreeTSD - free a Thread Swappable Data block

tkInitThrdKStack - Initialize Kernel Stack for a New Thread

tkInitThrdUStack - initialize user stack for a new thread

tkAllocTIB - allocate a TIB for the thread

tkSetTIB - set the thread's TIB

tkUpdateTIBPriority - set the thread's Priority in TIB

tkUpdateAllTIBs - force TIB update for all threads of a process

tklThrdRetError - panic return for internal thread

tkSuBuff386 - Copy from Resident System Memory to User Memory

tkSuBuff486 - Copy from Resident System Memory to User Memory

tkUsrMemError - Handle Fault in User Memory Routines

*****LP tkScanBuffz** - Scan a null-terminated (possibly two nulls) user buffer

tkScanBuffz scans a user buffer stopping either after the first NULL (2 NULLs) byte found, or after a specified number of bytes has been scanned, whichever comes first.

```
ENTRY:  tkScanBuffz(pLen, pMem, MaxLength, fDNULL);
        pLen = pointer to where length of buffer is returned
        pMem = address of buffer
        MaxLength = max. length to scan.
        fDNULL   = if TRUE scan for a double NULL

EXIT    NO_ERROR, if successful - Length is copied to pLen
        ERROR_TERMINATOR_NOT_FOUND
```

*****LP tkBufferWithinUserDSLimit** **;b723248**

```
*
*      This procedure returns error if the end of the passed buffer
*      is not within the user's DS selector limit.
*
*      ENTRY    Buffer      - base of the user buffer
*               BufSize    - size of the user buffer
*
*      EXIT     NO_ERROR, If successful
*               ERROR_PROTECTION_VIOLATION, otherwise
*
*      CALLS    KMQUERYCLIENTREGISTER
*
*/
```

```
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
;b723248
```

Tasking Local Procedures

Scheduler Design

The following section contains discussions of specific aspects of the design that are of technical interest to implementors and maintainers of the Scheduler component.

- Special handling of EFLAGS on kernel entry and exit

The contents of EFLAGS are not preserved across 16 or 32-bit API calls except for the TF, PM, MP, VM bits, Direction Flag and IOPL field. There are two complications relating to the restoration of the EFLAGS register, V86 mode and Single Stepping, which we will now discuss.

- Restoring EFLAGS when Single Stepping

For EFLAGS, care has to be taken to ensure that if the Single Step Flag (TF) was set in EFLAGS, a single step will be taken exactly after the return from ring 0. The basic approach here is to perform a POPFD just before the RETFD to user code. The POPFD instruction resets TF, but the TF bit is not acknowledged by the CPU until the end of the following instruction, i.e. immediately prior to executing the next user instruction.

If an external interrupt (NMI, processor extension segment overrun or INTR) occurs while single-stepping, the appropriate handler executes before the Single Step Interrupt handler (since these interrupts are processed after the Single Step Interrupt and the handlers are executed in the reverse order of processing). The Single Step Interrupt handler is then executed and, if a further single step debug command is input, the operation of restoring EFLAGS with TF set and returning to the user is performed by an IRET instruction. This means that the interrupted instruction is executed, TF is then acknowledged and the debugger is entered via the Single Step Interrupt handler.

- EFLAGS and V86 Mode

There is a complication relating to the EFLAGS register. The VM bit in that register is the method for distinguishing between the V86 mode and Protected Modes of the 80386 CPU. However, the VM bit in the EFLAGS register is cleared by the interrupt acknowledge or exception handling sequence. It is therefore essential to use the copy of EFLAGS saved by the sequence, not the value in EFLAGS after entering the kernel.

- Support for kernel stack overrun protection

The existence of user-written ring 0 code (device drivers and file system drivers) means that the usage of ring 0 stack is not entirely within control of the kernel. It is necessary to detect stack overrun exactly when it happens in order to prevent kernel data corruption and in order to assist in debugging the problem that caused the overrun in the first place. After-the-fact detection does not meet either of these goals; the damage is done and the culprit is hard to identify.

Catching the overrun when it happens means causing the offending instruction to fault. On the 386, there are two viable options:

- Use the segmentation hardware to catch the fault.

Map the kernel stack with an expand-down selector whose limit is set exactly to the base of the stack. Any attempt to access below the base through the SS selector will cause a stack fault (trap 0Ch).

- Use the paging hardware to catch the fault.

Place the base of the kernel stack at the bottom of a page of memory and make the page below not present. Any attempt to access within 4K below the base of the stack will cause a not-present page fault (trap 0Eh). Note that this method requires the base of the stack to be at the base of a page. Furthermore, a wild pointer reference that ventured more than 4K below the base of the stack might not be trapped.

Because of the restrictions and deficiencies noted with using the paging hardware to catch stack overruns, OS/2 v 2.0 uses the segmentation hardware even though this method is less portable.

- Critical sections protected by CLI/STI in PM

- Use A - Protecting critical sections entered at task-time from re-entry from interrupt time code.

Examples of this are device updates. The easiest way to protect this type of critical section is to prevent page faults occurring by locking down the relevant areas of PM code and data. Using an existing intermediate version of OS/2 as a basis for estimation, the amount of code and data to be locked down is approximately $(10k + 256n)$ bytes where 256 bytes is the size of a ring 2 stack and n is the number of ring 2 stacks (equivalently PM threads). Note that in the existing implementation of PM, the code that uses CLI/STI instructions to protect against interrupts calling the same code, and data is already locked in memory in order to create the ring 0 GDT alias selectors needed to access the code/data from interrupt level. The locking operation is performed by PMDD.SYS using the DevHelp_Lock function and specifying a long term lock. The segments of PM that are locked down are:

- DISPLAY.DLL - mouse pointer draw code and data

This consists of one 2932 byte code segment and one 2726 byte data segment

- PMWIN.DLL - hardware event queue code and data, and Fast Safe RAM semaphore code

This consists of one 1752 byte code segment and one 3076 byte data segment.

So a total of 10,486 bytes of memory are permanently locked down for PM, plus the ring 2 stacks.

- Use B - The implementation of Fast Safe Ram Semaphores in PM

A Fast Safe RAM Semaphore (FSRS) is basically an OS/2 RAM semaphore augmented with a process/thread owner field and use count. The following information is contained in the fast safe RAM semaphore data structure:

- 16 bit length field that specifies the length of the semaphore data structure, in bytes.
- 32 bit process/thread ID that uniquely identifies the thread of execution that owns the semaphore. If this field is zero then the semaphore is unowned.
- 16 bit client field that is not interpreted by the FSRS functions. Instead it is a field provided to the caller as a means for identifying which resource is currently owned by the owner of the semaphore. The client field is initialized to zero when a semaphore is first acquired, and thereafter the value of the field is under the control of the caller, until the semaphore is released. The client field is useful to a DosExitList handler in figuring out the appropriate cleanup action.
- 16 bit usage count. This count is incremented each time the current owner acquires the semaphore and decremented each time the current owner releases the semaphore. The current owner does not actually release its ownership until the count decrements to zero.
- 32 bit OS/2 RAM semaphore. This RAM semaphore is manipulated by the functions in this module in the same manner as the the ring 3 semaphore code in OS/2 (e.g. DOSSEMREQUEST and DOSSEMCLEAR).
- 32 bit time out value to use when waiting on the OS/2 RAM semaphore.

There is a reference count associated with each semaphore. In PM, the function that acquires a semaphore is called FSRSemEnter and the function that releases a semaphore is called FSRSemLeave. Every successful call to FSRSemEnter must be matched with a call to FSRSemLeave.

A FSRS combines the speed of a RAM semaphore with the safety provided by an OS/2 system semaphore. Speed is obtained by eliminating validation of pointers to FSRS data structures and by relying upon the caller to understand the FSRS data structure and perform the appropriate cleanup action in their DosExitList handler whenever a process dies. A further performance benefit is realized by only having to call the OS/2 semaphore request function if the semaphore is taken.

In order to implement fast safe RAM semaphores, one problem to be solved is that of grabbing a semaphore and marking it as being owned by the present process. This is needed so that PM can clean up any semaphores if a process dies while owning them. There are two parts to this problem, the first is taking the semaphore, the second to mark the PID of the new owner in it. Taking a semaphore is actually easy to do even if interrupts are enabled. All that is required is an atomic get-and-set instruction such as XCHG.

Taking a semaphore and recording its new owner is not so simple. In PM, semaphores are owned by threads, so the problem becomes that of storing a unique thread i.d. in the semaphore data structure when the semaphore is claimed. The combination of the process i.d. concatenated with the thread ordinal forms such a unique identifier. There is a critical section between the time when the semaphore is claimed and the time when the owner's i.d. is recorded during which the thread must not terminate, or else the semaphore structure will be left in an unrecoverable state. If the thread is preempted in this critical section, there is the danger that another thread in the process will run and cause the whole process to terminate. Under OS/2 v 1.2, it was sufficient for the thread to disable interrupts during this critical section. In a paged environment, however, there is always the possibility of a page fault occurring even while interrupts are disabled.

- Use C - Marking a PM object "busy"

The third use of critical sections in PM is the sequence by which PM validates an object and marks it "busy" to prevent simultaneous access to the object from other threads of a process. We need such a marking to make sure that no thread starts using an object while some other thread is deleting it. It would be bad, for example, if some thread was writing lots of drawing attributes into a device context object while a second thread deleted it, and maybe another process allocated the (shared) memory for some other object! Marking an object as busy takes two steps. First, we need to "validate" the object (e.g. verify that a set of device attributes is consistent for, say, drawing a line). Second, we need to mark it as busy to prevent other threads of the same process from accessing the object. But both of these must be done together as an atomic operation, since it does us no good to know that it was RECENTLY a valid object. Since a PM object may only be owned at a given time by one process and it must be owned before it can be accessed, the only concern is that another thread of the same process might have deleted or written to the object after one thread has validated it.

- Preventing page faults during critical sections

For new 32 bit code we specify three alternatives. The alternative to be used depends on the amount of code or data to be accessed while interrupts are disabled.

- Data will fit into the general registers

In this case, document that the data must be loaded into the general registers before disabling interrupts. To prevent page faults from occurring, code executed with interrupts disabled must not access memory, including the stack. Also, code must all reside on one page. This is guaranteed by knowing the total size of the code, and by using a page

alignment directive provided by the linker/masm. The typical case for which this is applicable is a short sequence of port updates using IN/OUT instructions.

- Not more than one page of code/data needs to be accessed

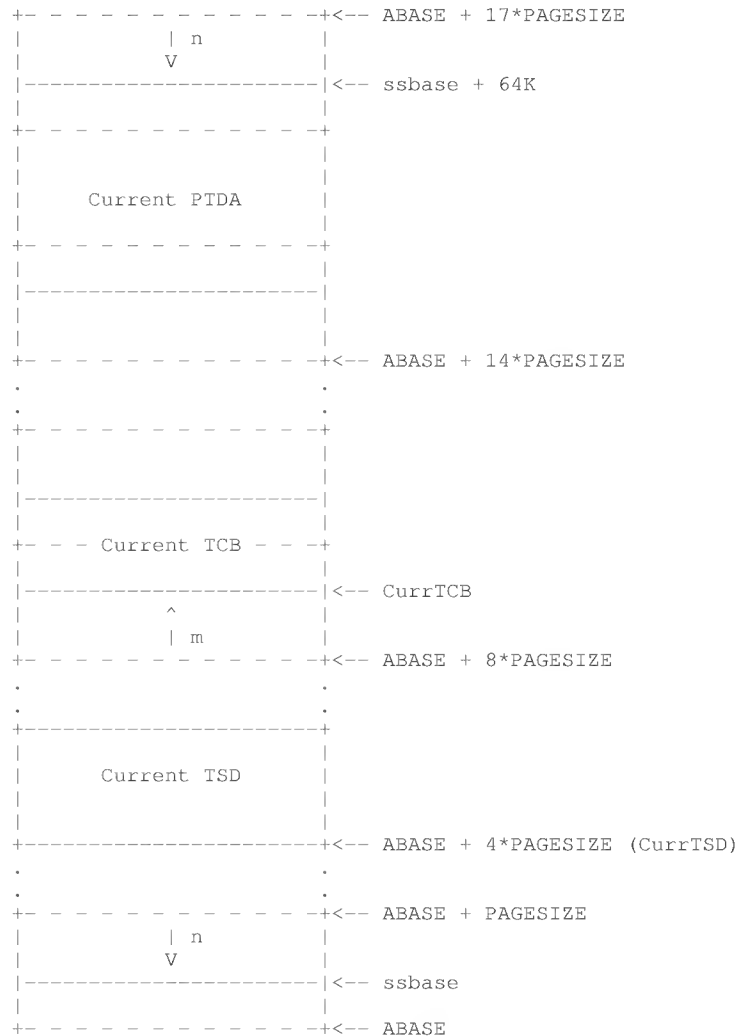
In this case, put all code and data to be accessed into a single physical page. Note that the page at CS:EIP is always guaranteed to be present.

- More than one page of code/data needs to be accessed

In this case a device driver must be used to lock the relevant code and data down. The cost of getting to the driver is probably acceptable for any situation with 4K+ of bytes to touch with interrupts off.

- Layout of region mapped by kernel SS.

When executing at task time at ring 0, the current thread's PTDA, TCB, and TSD are all addressable through SS. This is what the region mapped by SS looks like.



Kernel SS region layout

In the SS region layout, note that the PTDA may overlay as many as two page boundaries since it is larger than one page, while the TCB may overlap one boundary since it is smaller, and the TSD overlaps none since it is both smaller and page-aligned. The values needed for addressing within the SS region are computed using the following:

ABASE

The base of the alias region into which the current PTDA, TCB, and TSD are mapped (known).

CurrTCB

The offset relative to SS of the start of the TCB (computed).

CurrTSD

The offset relative to SS of the start of the TSD (computed). Note that this value minus one is the value of the limit field in the SS descriptor (which is marked expand-down).

m

n	The distance from the nearest page boundary below the TCB to the start of the TCB (computed).
PAGESIZE	The distance from the end of the PTDA to the nearest page boundary above (computed).
pPTDACur	The size of a page (known).
pTCBCur	The flat address of the current PTDA (known).
PTDASIZE	The flat address of the current TCB (known).
pTSDCur	The size of a PTDA (known).
ssbase	The flat address of the current TSD (known).
TKSSBase	The base value to place in the kernel SS descriptor (computed).
The computations are as follows:	
$n = (\text{PAGESIZE} - (\text{pPTDACur} + \text{PTDASIZE})) : (\text{PAGESIZE} - 1)$	
$\text{ssbase} = \text{ABASE} + \text{PAGESIZE} - n$	
$\begin{aligned} \text{CurrTSD} &= \text{ABASE} + 4 * \text{PAGESIZE} - \text{ssbase} \\ &= 3 * \text{PAGESIZE} + n \end{aligned}$	
$\text{TKSSBase} = \text{pTSDCur} - \text{CurrTSD}$	
$m = \text{pTCBCur} : (\text{PAGESIZE} - 1)$	
$\begin{aligned} \text{CurrTCB} &= \text{ABASE} + 8 * \text{PAGESIZE} + m - \text{ssbase} \\ &= 7 * \text{PAGESIZE} + m + n \end{aligned}$	

Implementation Estimates

The following are estimates for the lines of code (LOCs) that are affected. *These estimates are obsolete.*

Key:

LOCS = lines of code
 Effort = difficulty (0.5 = straightforward, 1.0 = average, 1.5 = moderately complex, 2.0 = difficult.
 Elocs = Effective LOCS = LOCS x Effort
 MM = Man Months, assuming 250 LOCs = 1 Man Month. A Man Month is an elapsed month assuming a regular 40hr working week.

	LOCS	Effort	Elocs	MM
NPX Manager	500	1.0	500	2.0
386 TSS	300	0.5	150	0.6
CPU Stepping Determination	50	1.0	50	0.2
Changes to kernel stack frame layout	100	0.5	50	0.2
Maps 32-bit call gate /IRET frame				
Accommodate 32-bit general registers				
Accommodate FS,GS segment registers				
Changes to dispatchers	200	1.0	200	0.8
Use 32-bit gates for all kernel entries	100	0.5	50	0.2
Call-gate emulation for 16-bit API that take an odd number of parameters				
B-bit clear for ring 3 stacks	25	1.0	25	0.1
Must zeroise upper word of ESP when returning to ring 3 if B-bit clear.				
B and G-bits clear for thread r0 stacks	50	1.0	50	0.2
Must zeroise bits 16-31 of ESP on every entry to the kernel.				
Changes to the Libi_Dispatcher routine	50	1.0	50	0.2
Must handle both 286 and 386 EXE format				

DLL's.				
Exitlist on r3 stack with B-bit set	50	1.0	50	0.2
If thread 1's ring 3 stack has the B-bit set we must use an alias selector to a portion of that stack not exceeding 64K bytes so that 16-bit API calls can be made from ExitList routines.				
[This design not finalized]				
Sig handlers on r3 stack with B-bit set	25	1.0	25	0.1
Dispatch to signal handlers must be delayed if thread 1's ring 3 stack has the B-bit set. Upper part of ESP trashed if 16-bit gate into r2 called from signal handler.				
[This design not finalized]				
Modifications to Context Switching	50	1.0	50	0.2
Need to remap current PTDA and LDT by editing GDT descriptor base and limit fields.				
Changes to CreatePTDA	25	1.0	25	0.1
Forge PTDA links by linear address				
Change to calling TKInitTCB				
TKInitTCB, linear version of	25	1.0	25	0.1
Init_TCB_Stack				
Accept linear execution address				
Accept linear PTDA address				
Accept linear TCB address				
TKProcAlloc, linear version of	25	1.0	25	0.1
ProcessAllocate				
Accept TCB linear address. Use it to link TCB into Scheduler. Callers of ProcessAllocate change to using TKProcAlloc				
Changes to w_CreateThread	50	1.0	50	0.2
Merge GetTCB into w_CreateThread				
Don't grow PTDA to create a new TCB. Instead grab TCB's off communal free list and error if none available. Use linear address to access and link the new TCB.				
Change to calling TKInitTCB				
Don't initialize Stack Fence				
TKTidToTn, linear version of TidToTn	25	1.0	25	0.1
Specify target PTDA linear address instead of PTDA selector.				
Thunk TidToTn calling TKTidToTn.				
TKScanThreads, linear version of	200	1.0	200	0.8
EachThread				
Specify target PTDA linear address instead of PTDA selector. Scan of ActiveList uses linear addresses of TCB's in place of PTDA offsets of TCB's. Specify 32-bit worker routine linear address. Must convert callers of EachThread to call TKScanThreads.				
TKTnToTCB, linear version of TnToTCB	100	1.0	100	0.4
Returns TCB linear address instead of offset from PTDA. Callers must be converted to use TKTnToTCB.				
TKTidToTCB, linear version of TidToTCB	100	1.0	100	0.4
Returns TCB linear address instead of offset from PTDA. Callers must be converted to use TKTidToTCB.				
TKDescendant, linear version of	25	1.0	25	0.1
Descendant?				
Thunk Descendant calling TKDescendant.				
SIGResetFocus, linear version of	25	1.0	25	0.1
ResetSigFocus				
Thunk ResetSigFocus calling SIGResetFocus.				
SIGPost, linear version of POST_SIGNAL	25	1.0	25	0.1
Thunk POST_SIGNAL calling SIGPost				
SIGCurrentPending, linear version of	25	1.0	25	0.1
SIGActive				
Thunk SIGActive calling SIGCurrentPending				
TKScanTasks, linear version of	200	1.0	200	0.8
ScanTasks				
Accept 32-bit worker linear address. Scanning uses linear address PTDA links. Worker called with linear				

address of PTDA. Existing ScanTasks /EachTask interfaces retained in the short term. Cannot use thunks to call TKScanTasks.				
TKPidToPTDA, linear version of PidToPtda	25	1.0	25	0.1
Thunk PidToPTDA calling TKPidToPTDA.				
TKLinkTask, linear version of LinkTask	25	1.0	25	0.1
Forge both linear and selector based PTDA links. Callers of LinkTask change to using TKLinkTask.				
TKUnLinkTask, linear version of UnLinkTask	25	1.0	25	0.1
Break both linear and selector based PTDA links. Callers of UnLinkTask change to using TKUnLinkTask.				
TKCWait, linear version of CWait	25	1.0	25	0.1
Callers of CWait change to using TKCWait.				
TKCollectZomb, linear version of CollectZomb	25	1.0	25	0.1
Update linear zombie queue links. Callers of CollectZomb change to using TKCollectZomb.				
TKNewPid, linear version of NewPid	25	1.0	25	0.1
Callers of NewPid change to using TKNewPid				
TKDescendant, linear version of Descendant?	25	1.0	25	0.1
Thunk Descendant? calling TKDescendant.				
TKAdoptTask, linear version of AdoptTask	25	1.0	25	0.1
Change to using linear address PTDA links. Callers of AdoptTask change to using TKAdoptTask.				
TKBlock, linear version of ProcBlock	50	1.0	50	0.2
Single 32-bit parameter for block-id				
Manipulate block-id's in single PblkKey array. Thunk ProcBlock calling TKBlock.				
TKRun, linear version of ProcRun	50	1.0	50	0.2
Single 32-bit parameter for block-id				
Manipulate block-id's in single PblkKey array. Thunk ProcRun calling TKRun.				
TKForceThread, linear version of ForceThread	25	1.0	25	0.1
Accept linear TCB address. Callers of ForceThread must change to using TKForceThread				
Mods to Set_Force, Quiet_Set_Force	25	1.0	25	0.1
Modify ActiveList scanning to use linear TCB Addresses.				
ActiveList scanning outside TK	100	1.0	100	0.4
Mainly 3xBox, Semaphores, Loader, Ptrace.				
Replace 52 references to built-in TCB	200	1.0	200	0.8
52 references to offset TCBPtr to access thread 1's TCB need to be replaced since that TCB no longer lives in the PTDA. Assumes 4 LOCs ea.				
Page Faults when interrupts off	50	1.0	50	0.2
Intra-process thread switches disabled by page fault handler if entered when interrupts off.				
Page Faults during stack switches	50	1.0	50	0.2
Internal modifications to Page Fault, Trap 1 and Trap 3 handler to prevent signal dispatch while user stack is inconsistent.				
Subtotal	3100		2850	11.4

Glossary

0:32	Refers to a flat pointer consisting of a 32-bit offset relative to the beginning of the 80386's four gigabyte virtual address space.
16:16	Refers to a segmented pointer consisting of a 16-bit selector value and a 16-bit offset relative to the segment that the selector selects.
Ager	An internal thread owned by the Page Manager that is responsible for causing the contents of pages of physical storage that have not been used recently to be written out to the swap device so that the page of memory may be reused. This is a very simplistic explanation.
BMP	Block Management Package. A package for managing records of fixed size provided by the VMM.
DosLibInit	Undocumented entry into the kernel by which the LibInit dispatcher at ring 3 passes control back to the kernel to set up for dispatch to the next LibInit routine.
GetNextRunner	The entry point in the Scheduler that selects the next thread to be given the CPU.
SCI	System Call Interpreter. A threaded interpreter used by 16-bit kernel APIs to validate parameters and load them into registers, to invoke the API worker routine, and to write return values back out to user space. The idea was that by using an interpreter to perform tasks common to many APIs, code space in the kernel could be saved.
slot number	Each thread in the system has associated with it a number which is its index, or slot, in various tables. This slot number serves as a global identifier for the thread within the kernel. Slot number zero is reserved.
starvation	A thread whose priority class is "regular" receives a priority boost whenever the system determines that it is runnable but has not been run in a reasonable length of time. Such a thread is said to be starving, and starvation refers to keeping track of which threads are starving and which are not.
thop	A threaded interpreter opcode. SCI interprets a stream of thops. The term is also used sometimes to mean the procedure that implements the opcode.
VDM	Virtual DOS Machine.

User-mode Debugging

This page is intentionally left blank.

Debug Architecture

This page is intentionally left blank.

Debug Problem Description/Objectives

Debug consists of a set of functions that provide the features necessary to locate and correct problems in programs, to speed the program development process.

Main Objectives of Debug

- Debug should provide the minimum required elements to build excellent debuggers, providing a debugging interface that is
 - Fast
 - Powerful
 - Flexible
 - Non-Intrusive
 - Accurate
 - As Portable as possible
 - Not a performance burden
 - Capable of debugging Programs and DLLs
 - Capable of debugging mixed or pure 16 and 32 bit applications
 - Debug should be restricted from intruding on
 - Kernel code, data, or registers
 - Ring 0 Device Drivers
 - Non-Debugged Processes
 - Debug must coexist with the kernel debugger and the ICE 386.
-

Considerations In The Design of Debug

- Audience

Any person who writes software for OS/2 could potentially use Debug.

To a software developer, having a fast, powerful debugging tool is very important. The CodeView/P debugger is viewed as important to the success of all the language products that we produce. Debug is the key functional element that allows CodeView/P to work in OS/2.

Debug would not be useful in an office environment.

The types of programs that would normally use Debug include

 - Interactive Debuggers
 - Automated Profilers
 - Automated Testcase Drivers
- Performance

The pattern of use for Debug is of long-term disuse, and short-term intensive use. Therefore, Debug should be optimized for the

case where it is not in use, but still be responsive when in use.

Debuggers may wish to single-step the debuggee repeatedly, while doing a complex software watchpoint operation. This means that the single-step operation, memory reads, register reads, and thread status becomes a bottleneck. This software watchpoint scenario and memory access speed are the main driving forces to improve Debug performance.

An interactive symbolic debugger, such as CodeView/P, spends so much time looking up symbols or doing I/O, that Debug is rarely a performance bottleneck.

Interactive Debuggers and Automated Testcase Drivers do not have an extreme need for performance. While higher performance is always desirable, the key to these utilities is powerful, non-intrusive debugging, rather than speed.

Automated Profilers need the best performance possible. They may wish to run the debuggee at near-normal speed, while single-stepping each instruction.

Because of the resolution of the system's clock, and preemptive scheduling, Debug is not a very accurate method to time program execution. Timer hardware, profiling kernel support, or RAS would be better suited for this. Debug is useful to profile program execution paths, but not to time them.

When Debug is not running (or has never been run) there should be no interference caused by its presence in the system. Debug should require no 'background' activity when not in use.

- Protection

Protection becomes an important factor when granting a process the power to modify the registers and memory space of another process, especially when granting the option to modify normally-constant memory regions, such as code and read-only data.

In the context of Debug, protection means to keep system interference to a minimum when debugging. This means preventing memory accesses from accidentally corrupting the kernel, DLLs, or other processes. This isolation is especially important when considering that many memory modifications result directly from people experimenting with their code or data.

Protection cannot be made perfect and still maintain a useful function. Literally, the only perfectly protected system is one that does not run applications. So, Debug needs to strike a balance between protecting other processes from a debugger, and the usefulness of debugging.

Generally, protection in Debug takes the form of limiting debugger memory accesses to a harmless memory space, and assuring that normally constant memory modified by the debugger is private to the debuggee.

- Usability

The code that most applications programmers write for OS/2 consists of

- Batch Files (foo.BAT and foo.CMD)
- Real Mode Programs
- Protect Mode Programs (Processes and Threads)
- Libraries (DLLs)
- Ring 0 Device Drivers

Debug is not intended for use with Batch Files, as the format of these is unknown to the OS/2 kernel. Debug can not access any object at ring 0, including ring 0 device drivers. The Kernel Debugger or RAS components are better suited to these debugging problems. Debug is not currently intended for assisting real-mode development. Existing real-mode debuggers should be used for real-mode debugging. Debug will not interfere with real-mode debugging.

Debug provides debugging connections on a per-process basis. These connections are independent of screen groups, so that even a detached or synchronous process may be debugged, via indirect DosExecPgm calls. A debugger may not debug itself, nor any process that is debugging it. However, a debugger may debug a copy of itself.

A debugger may debug more than one process simultaneously, but any debuggee must have at most one debugger. In general, this set of debugging connections forms a collection of trees.

Threads within processes are the main objects that Debug works with. DLLs do not have threads associated with them directly, but are accessed and manipulated by threads of client processes. So, Debug supports debugging of DLLs by permitting the debugger to observe the memory of the DLL, and the registers of a client thread while it uses the DLL.

Debug also assists the debugger in debugging DLLs by telling the debugger when a library is loaded. The debugger may also use Debug to get the name of a library, or to convert a logical segment number and MTE to an address. By storing these items, debuggers can transform an address or value to a symbolic reference.

Debugging connections are designed to be consistent on a per-processor-architecture basis, but can not be perfectly portable from one machine to the next. When new features are added to Debug to support a new machine, a new debugging level will be added, with the possibility of supporting translation between levels for backwards compatibility. The initial command must establish that the debugger application and Debug understand each other. Programs which use Debug may not be perfectly portable, and may need to be changed to work on new versions of OS/2.

- Flexibility

Debug should provide the minimum required functionality that is required to debug programs, leaving the debugger application to use these functions to create a usable debugger. By using many repeated calls to Debug, and saving required information, a debugger may be customized to the user's whim.

Debug should not have to worry about things such as how I/O is accomplished, or the symbolic meanings of various objects in the debuggee. This way, a debugger may be implemented to work with any of the computer languages. It may also be constructed so that its commands come from a keyboard, are generated automatically, or come from any form of IPC, including a network.

Note that the debugger may actually capture and use the thread and memory space of the debuggee for its own purposes. In other words, debugging stubs can be incorporated into the debuggee process itself, and that thread can execute these stubs under the complete control of the debugger. This technique can be used by the debugger to create and execute Trojan-Horse routines that would be impossible in the context of the debugger process.

For example, these functions could :

- set the priority any thread of a debuggee
- signal a debuggee's child subtree
- access a debuggee's file, opened as 'deny-all'

This can also be used to execute user-defined functions, which would perform some debugging stub task, such as printing a list entry using the debuggee stub procedure to do the formatting, and the debugger to do the I/O.

- **Compatibility**

It would be good to maintain the old-style debugging interface, exactly like PTRACE was done on the 286 system, to protect the investment in current debuggers. However, this is not possible. The PTRACE interface can not distinguish between 16-bit and 32-bit code or data segments. So, if allowed to debug a 32-bit segment, no purely 16:16 debugger could possibly disassemble code correctly. This means that old debuggers will not be able to debug old code with the PTRACE interface.

So, the old PTRACE interface will be scrapped. That is, the PTRACE ordinal will fail to link to any function. This way, if a debugger attempts to link to the old PTRACE interface, the debugger will fail to load.

A new interface, called Debug, will replace the old PTRACE interface. Debug will attempt to create a superset of the PTRACE functions, and the command and buffer formats will be similar to PTRACE. However, the interfaces will not be identical.

- **Portability**

Debug is an inherently non-portable function, because of the differing register sets between processors.

Debug should be created so that the machine-dependent functionality, such as the format of register sets, is separated from the machine-independent functionality, such as the logic of the Go or Terminate commands.

Debugger authors should be aware that the Debug function may behave differently on different systems, and that programs that use Debug may not be compatible with all future versions of the system.

- **Debugging Power**

Debugging power is a general term used to refer to the number and usefulness of basic tools available from a debugger, and the classes of problems that can be solved easily. For example, a debugger with a data watchpoint capability is more powerful (other things being equal) than one without.

Debug should provide a basic set of debugging functions. Debug should exploit the range of debugging aids available on the processor as fully as possible. It is up to the debugger to combine these functions to create a powerful debugger.

Debug should provide an interface that solves just those problems that are impossible to solve at the application level, leaving the rest up to the debugger.

Debug supports functions such as single-stepping, register and memory modifications, while the debugger application supports breakpoints, symbols, languages, and the user interface.

- **Debugging Intrusion and Limitations**

A debugger should be accurate, showing as closely as possible the operations that a thread or process executes.

However, there are always limitations that complicate the situation. For instance, if a debugger is in use, the timing of scheduling and semaphores is affected.

Note that intrusion may become a very serious matter. The most serious known problem comes when a debuggee requests and sets a semaphore, then hits a breakpoint. If the debugger requests this same semaphore, without letting the debuggee clear it, a deadlock will occur. This can happen with any semaphore shared between the debugger and debuggee, including semaphores not implemented by the OS/2 base. This becomes most serious when it happens to a semaphore vital to the operation of the system, such as the VIO semaphore.

A kernel solution to this problem is as yet unknown. So, it is the responsibility of the debugger author to see to it that this intrusion, especially deadlocks, is minimized.

Debug Functionality Selection

- Wishlist Items

A number of wishlist items have arisen by using the existing Debug function. Below is a list and analysis of each wishlist item, followed by a selection of the items which will be included in the Debug function.

1. Get selector information (limit, access rights, MTE)

This idea is to pass Debug a selector number, so that debuggers can display a list of debuggee segments. The MTE is valuable so that the debugger can load symbols for that DLL.

This can all be done in the current 1.1 implementation. The debugger can use the Trojan-Horse technique to get the limits and access rights. The MTE is told to the debugger via the LibLoad event.

However, this function is crucial to debugging in the 386 environment. A debugger needs to know the value of the D-bit or the B-bit in all segments for disassembly. Knowing the limit and access rights is especially useful for the current set of segment registers, especially CS and SS, for both disassembly and understanding faults.

This function is not portable to other architectures. Because of this, the limit and access rights for each segment register should be returned as a part of the standard 386 register set, and should be updated in step with the segment registers. For protection, the actual limit and access rights may never be modified by writing the registers.

2. Get Semaphore Status

When writing programs for a multitasking system, the use of Semaphores is often essential. However, misuse of these semaphores often results in deadlocks, or timing bugs that are difficult to locate. Often, a number of different semaphores are in use at once, and their interrelationship can be extremely difficult to understand.

To aid in locating timing bugs or deadlocks, it can be valuable to know the state of semaphores at any particular time. A Graph of the semaphore sets, clears, and requests can be extremely valuable in determining race and deadlock conditions.

Because this is interesting for debugging, it has been suggested that a Semaphore Status command be added to Debug. This command would take a thread ID and a buffer pointer as input. If the thread is blocked, Debug would return the semaphore address, name, and type.

However, starting with OS/2 release 1.2, the DosQueryProcStat API will be available. This API returns status information of processes currently running in the system. Included in this status information is exactly the information that the Semaphore Status command would normally return.

Because this functionality is already available, there is no need to add it to the Debug component also. Debugger applications which wish to peruse semaphores should simply use DosQueryProcStat.

3. Deadlock Correction Function

A deadlock can occur due to the debugging architecture we have chosen to support. The deadlock occurs if the debuggee sets a shared semaphore (of any kind, often the VIO semaphore), then hits a breakpoint. If the debugger then requests this semaphore, a deadlock occurs.

One 'deadlock-correction' function would be something like this

```
For (each debuggee thread) {
    While (it owns a dangerous semaphore) {
        Single-Step that thread
    }
}
```

The hard part is determining what a 'dangerous' semaphore is. This is where the Get Semaphore Status function might be of some use.

The key to solving this problem is to keep the debugger from requesting semaphores at the same time the debuggee does. Naturally, this can be very difficult for an interactive debugger.

One possible solution is to have the debugger know the 'danger areas' in the debuggee code, and single-step through those areas without requesting the 'dangerous' semaphore.

Another solution is to use two machines linked by some safe communications medium, such as a serial port. The user interface resides on one machine, while a stub debugger resides on the machine with the debuggee. The stub debugger would use only Debug and the safe communications medium to do its work, never requesting a 'dangerous' semaphore.

However, because this scenario is deeply rooted in the debugging architecture, no amount of effort by the Debug component can correct this problem. It is up to debugger applications to assure that this does not happen.

4. 80386 register read/write

In order to support the 32 bit architecture of the 386, a new function must be added to Debug to read and write the registers of the 386. This is a crucial operation; without this, Debug would be worthless.

5. 80387 NPX Coprocessor register read/write

Intel's 80387 NPX has a slightly altered context buffer from the 80287. In order to ramp-up from the 287 to the 387, new commands should be added to Debug to support this. The 287 Debug commands are simply not sufficient for this.

Additional Coprocessors are bound to be added to the machines that run OS/2 as time goes by. For this reason, the commands to read and write the coprocessor registers should also provide a path for portability to new coprocessors as needed.

6. 80386 debug register control (Data Watchpoints)

Intel's 80386 CPU has a set of debugging breakpoint registers that provide debuggers the ability to set code and data breakpoints. This extremely useful hardware function should be exploited to its fullest, to provide a large increment in debugging power. A set of commands will be added to Debug to manage these registers.

However, we do not want to needlessly tie Debug to the format of these debug registers. Debuggers use debug registers for only one purpose, and that is to set watchpoints. So, Debug should have watchpoint commands, which happen to be implemented using the 386 debug registers. This way, if OS/2 is ported to a machine with a different debug register format, the watchpoint interface can still be supported, largely unchanged.

The watchpoint commands would include a command to set a watchpoint, another to clear a watchpoint, and a watchpoint hit notification.

7. Better Control of Exceptions

Today, a GP fault always behaves differently when being debugged than when not being debugged. Sometimes, it is desirable to let the system go ahead and kill the process, rather than intruding on the environment. Providing better control of exceptions means giving the debugger the option of selecting the standard action for the exception, or being notified if the exception occurs.

This cannot be done any other way than by adding the function to Debug.

8. Notification of more Exceptions

Today, the set of exceptions that a debugger can know about is very limited, with only traps 01, 03, and 0D being available. Debuggers would like to see more exceptions, such as divide-error, overflow, and others, for better debugging.

Both being able to selectively control exception notification, and notification of more exceptions should be supported at the same time, as each one is largely dependent on the other.

Any other hardware implementation will most likely have a different set of exception events, which will be defined later.

9. Remote Debugging

This means having a debugger that functions across a network.

The IPC can all happen via the network of the user's choice, communicating with a slave debugger, which then in turn debugs the target application(s) on the slave machine. This scenario is independent of the Debug implementation.

Because of this, Debug will never need to function across machines.

If the question of portability is addressed properly, the Debug interface should be flexible enough to be emulated over any kind of IPC. In other words, if the Debug buffer format is designed correctly, it should be usable as the base of an IPC protocol (the base of the data packet format) for debugging.

10. Subsystem (DLL) Debugging

This is supported today, by allowing a debugger to read and modify DLL segments.

It is up to debuggers to derive the power of subsystem debugging from Debug.

The idea of Protected Dynlink Libraries is currently being discussed, where the memory space would expand and

contract on a per-thread basis. The impact of this on the Debug read/write memory commands will be discussed at a later time, when that memory model is more fully understood.

11. Trojan Horse debugger enhancements

This is supported today. A debugger can modify a debuggee's code and data segments at will, and completely control its registers. The debugger can create routines in the debuggee's address space, and use Debug to run them. In this way, the debugger can use the full set of API functions to read or modify the debuggee's environment.

This is useful for extending a debugger's functionality as far as desired, and can even be used to let the user write customized debugging routines. This option should be utilized by debuggers whenever possible, rather than adding functionality to Debug, as adding functionality adds size to the OS/2 kernel.

12. Debugging V86 or Real mode programs with a Protect mode debugger

By doing this, we virtually eliminate the memory overhead that occurs by having a debugger running below the 640K barrier. This is very desirable for debugging real and V86 mode applications.

This is a useful function if we decide that we want to use OS/2 as a development platform for DOS applications. It is fairly clear, because of the installed MS-DOS base, and the fact that there are a number of machines still being designed and produced that cannot run OS/2, that MS-DOS will be around for quite a while after OS/2 is available. Utilizing OS/2 as a development platform for DOS seems reasonable, in this case.

The problem remains as to how to enable debugging on the target application, as the debugger possibly may not be able to start the application directly.

There are a number of technical difficulties that arise if the V86 mode is not used for emulating real mode. Some of these include intercepting exception vectors correctly, and the fact that the real mode box is frozen when not the foreground screen group. For this reason, This function is dependent on both MVDM and 32-bit Debug availability.

13. Peeking into Pipes and Queues

This cannot be supported directly from Debug. Pipes are implemented internally to the kernel, and Queues are implemented entirely within a DLL. These two pieces of functionality are representative of generic debugging problems that Debug cannot address directly.

a. Breakpoints on API calls

Debug is not designed to provide any access to the OS/2 kernel code, data, or registers. By allowing a debugger to place a breakpoint at the ring 0 entry point to an API call, Debug would break this rule.

Also, other processes may wish to use the API, and when they make the call, they would hit an unrecoverable breakpoint.

Debug may set a breakpoint in the DosCall1 library, because that not ring 0 code, and it is made private when the breakpoint is placed there. No debugger should ever have a direct dependency on the format of this, or any other, system library, as the implementation of the libraries may change between versions of OS/2.

b. General DLL Symbolic Debugging

Queues are implemented completely within a DLL library. DLLs were designed with this class of module in mind - ones with hidden data structures and a well-defined interface.

Debug is unable to discern the structures that DLL authors use internally. It is up to the debugger application and the DLL author to work together to ascertain this information.

Debug does provide the register and memory access needed to implement this feature. A DLL author can construct a 'peek' debugging stub for this use. When the debugger wishes to examine the data structure, it can cause the debugged thread to run this stub, and then capture the stub's output for further formatting and printing. In this way, any DLL data structure can be examined - not just Queues.

14. Capture a running application and convert it to being debugged.

This is an architecture unsound idea, as it would violate the protection provided by OS/2, without a significant offset in utility, even though it seems technically feasible. A debugger must have identified a process as a debuggee before running it.

This option should be avoided.

15. Connect To Debuggee command

This command would be added purely to enable Debug to smoothly port from one machine to the next. The command is required to be the initial command for a particular debugging connection. It defines the set of commands and the

buffer format available to the debugger for the rest of the debugging connection. This set of commands is identified by the debugging connection level, passed in with the command.

This command is to be machine-independent, with the exact same buffer format from one machine to the next. This is accomplished by only reading the first few fields of the Debug buffer, for this command only.

Once set, the debugging connection level may never be changed.

If Debug is ported to another machine, it is very likely that the old set of commands and the buffer format would not be sufficient for the new architecture. This command would tell the debugger that the particular level is or is not available, and the debugger can act accordingly from then on.

For architectures where the new commands are a simple superset of the old commands, a new debugging level will still be added, but the old one may also be available via internal translation. For this reason, debuggers which are able to debug on more than one level should attempt the connection at the most complex known level first, and progress backwards to the first available level. This should tend to maximize the debugging power available to a multi-level debugger.

It is conceivable that a very smart debugger would be able to understand any of a number of different machines at once. In this case, the debugger would be able to attempt to set each different level that it knew of, until one was accepted. Typically, a debugger application will only be able to handle one level of buffer format at one time, and if that level is not supported, the debugger will not be usable.

A command that queries the native debugging format of the debuggee would not be required because nearly all debuggers would be written to support just one format, or at least a very small number of formats. In both cases, a Query function is not required, and simple polling is sufficient.

If the debugging level is not available, the debugging connection will not be made, so that another debugger process may be started to establish a more specialized debugging connection.

16. Buffered Read and Write Memory Command

This command would pass a debuggee address, a debugger address, and a length, and would transfer a copy of that number of bytes from the debuggee's memory to or from the debugger's memory, starting at the specified addresses.

The requirement for this command comes from the 0:32 address space that we have decided to support for the 80386 processor. To implement memory aliasing, this memory model requires that aliases be done at the linear level.

The difference between read-only and read-write linear memory is controlled by a bit in the page table entry. On the 80386, a read-only page is read-only at ring 3, but is read-write at all other ring levels. So, because a debugger may execute at ring 2, all linear memory aliases must be treated as having read-write permission.

This implies that whenever a memory alias is requested via Debug, the aliased areas must be converted to private memory, so that memory writes to constant areas do not interfere with other non-debugged processes.

Read-only memory aliases were provided with the explicit caveat that they never allocated memory, so that debuggers could simply examine memory without consuming it. Because it is expected that debuggers are going to read many of the segments in a debuggee process, this could have disastrous results to the memory used by the system. An out-of-memory error could occur while a debugger was simply disassembling code. The logic in current debuggers would not be able to handle this error, as it would mean terminating the debuggee process to recover the memory. This out-of-memory condition would tend to occur mainly on the largest of debugging sessions, where having a fatal error such as this could possibly ruin days worth of difficult work.

The buffered read memory command would provide a fast memory access alternative to read-only aliases in cases where the read-only aliases can not be supported.

It is expected that Intel will eventually alter the meaning of this read-only bit to become truly read-only on some future processor, or perhaps even in future steppings of the 80386. For this reason, the interface to map a read-only alias will be defined, but will return errors when it cannot be supported.

Because the buffered read memory command is a hard requirement, a buffered write memory command will also be added to provide a clean debugging interface, even though the buffered write memory command is not a hard requirement.

- Wishlist Ranking

- Possible today - no new Debug functionality

- Remote Debugging (across a network or phone line)
 - Subsystem (DLL) Debugging
 - Trojan Horse debugger enhancements
 - Debugging Stubs
 - Get segment information (limit, access byte, MTE)
 - Semaphore Status (using DosQueryProcStat)

- These MUST be included in the 386 enhancements
 - 80386 register read/write
 - Coprocessor register read/write
 - Watchpoint control
 - Access to D-bit and B-bit in access rights of CS and SS selectors
 - Connect To Debuggee command
 - Buffered Read/Write Memory commands
 - Good candidates for 386 enhancement
 - Control of exceptions
 - Notification of additional exceptions
 - Explicit segment information (limit, access rights)
 - Good ideas, but must wait
 - Debugging V86 mode from Protect mode
 - Technically Impossible or Unsound
 - Accessing any System (Ring 0) object.
 - Breakpoints on API calls.
 - Capture a running application to be debugged.
 - Peeking into Pipes
- Proposed New 386 Debug Functionality List
 - 80386 register read/write
 - Explicit segment information (limit, access rights)
 - Coprocessor register read/write
 - Watchpoint control
 - Notification of additional exceptions
 - Control of exception notifications
 - Connect To debuggee command
 - Buffered read/write memory commands

Solutions/Justification

This section defines the user interface for debugging on the 386.

The old PTRACE call for the 286 environment will not be supported. The old interface is not sufficient for debugging in a 32-bit environment. Due to the mixed 16 bit and 32 bit environment, the debugger cannot reliably disassemble 386 code segments without knowing the D-bit in the CS selector and the B-bit in the SS selector

This means that old-style debuggers will not be able to debug old-style applications on the 386 system.

The command and notification numbers for PTRACE and Debug will be similar, but will not be completely compatible. For example, the ReadReg command uses the same command number constant for both systems, but the register set is vastly different.

- Naming Conventions

The Debug command numbers and notification numbers are be given a new prefix to make them more understandable, and to distinguish them from the PTRACE command numbers.

286 version (PTRACE)

TRC_C_Xxxx	means	Trace - Command - Xxxx
TRC_C_XXX_ret	means	Trace - Constant - XXX - returned

386 version (Debug)

DBG_C_Xxxx	means	Debug - Command - Xxxx
DBG_N_Xxxx	means	Debug - Notification - Xxxx

Command and Notification Parameter Usage Diagram (simplified)

```
(                                ) = DBG_C_Null ()
(Value                          ) = DBG_C_ReadMem (Addr)
(                               ) = DBG_C_WriteMem (Addr, Value)
(Regs                          ) = DBG_C_ReadReg (Tid)
(Regs                          ) = DBG_C_WriteReg (Tid, Regs)
(                               ) = DBG_C_Go ()
(                               ) = DBG_C_Term ()
(                               ) = DBG_C_SStep (Tid)
(                               ) = DBG_C_Stop ()
(                               ) = DBG_C_Freeze (Tid)
(                               ) = DBG_C_Resume (Tid)
(Addr                          ) = DBG_C_NumToAddr (Value, MTE)
(*Buffer                       ) = DBG_C_ReadCoRegs (Tid, Value, Buffer, Len, Index)
(*Buffer                       ) = DBG_C_WriteCoRegs (Tid, Value, Buffer, Len, Index)
(*Buffer                       ) = DBG_C_ThrdStat (Tid, Buffer)
(Buffer                        ) = DBG_C_MapROAlias (Addr, Len)
(Buffer                        ) = DBG_C_MapRWAlias (Addr, Len)
(                               ) = DBG_C_UnMapAlias (Buffer)
(                               ) = DBG_C_Connect (Pid, Tid, Value)
(*Buffer                       ) = DBG_C_ReadMemBuf (Addr, Buffer, Len)
(                               ) = DBG_C_WriteMemBuf (Addr, Buffer, Len)
(Index                         ) = DBG_C_SetWatch (Value, Addr, Len)
(                               ) = DBG_C_ClearWatch (Index)
(                               ) = DBG_C_RangeStep (Tid, Value, Addr)
(                               ) = DBG_C_HandleException (Tid, Value)
(Value, MTE, Buffer, Len) = DBG_C_AddrToObject (Addr)
-----

(                                ) = DBG_N_Success
(Value                          ) = DBG_N_Error
(Value                          ) = DBG_N_Signal
(Addr                          ) = DBG_N_SStep
(Addr                          ) = DBG_N_Breakpoint
(Value, Index                  ) = DBG_N_ProcTerm
(Value, Addr, Len, Index       ) = DBG_N_Exception
(Value                         ) = DBG_N_ModuleLoad
(Value                         ) = DBG_N_CoError
(Value                         ) = DBG_N_ThreadTerm
(                               ) = DBG_N_AsyncStop
(Value                         ) = DBG_N_NewProc
(Buffer                        ) = DBG_N_AliasFree
(Value, Addr, Len, MTE, Index  ) = DBG_N_Watchpoint
(                               ) = DBG_N_ThreadCreate
(Value                         ) = DBG_N_ModuleFree
(Value, Addr                   ) = DBG_N_RangeStep
```

Architectural Review

The initial architectural review identified some problems in the initial design. Most of these were trivial, and are not addressed below. However, the question of portability was not considered, and the initial design showed some very serious flaws in this light. A complete review of this design was required to account for portability.

Question from Pete Stewart :

Can we provide support for non-portable features in a portable manner, such as debug registers ?

Reply :

This was not considered in the initial design of the 386 PTRACE. However, it certainly should have been. This is such an important issue that it requires a review of the entire document, and extensive changes. This portability concern resulted in the following changes.

- The debug register commands were abstracted into watchpoint commands, to hide the internals of the debug registers.
- The command numbers and exception numbers were changed to be compatible with the 286 numbers, with some extensions. All future versions of Debug are expected to use this same technique to minimize confusion. The command numbers are also compatible with the existing Xenix numbers (have the same meaning), with extensions.
- The new exception notifications were each given their own notification numbers, so that the old notification numbers

were still usable.

- The ExEnable and ExDisable commands were renamed NEnable and NDisable, and now take notification numbers as input, rather than exception numbers. This gives the ability to disable other notifications in the future.
- A command was added to establish the debugging connection, with a parameter that tells the connection level. This command can be extended to new processors as needed.
- The NPX commands were condensed into a single pair of commands, with a parameter that specifies the coprocessor type. This was done so that we can support new coprocessors with a minimum of trouble.

Another major change was the transformation from the 16:32 to the 0:32 memory model. Moving to a flat architecture resulted in the following changes.

- The read-only aliases were scrapped for the 80386. This is because the read-only bit in the page table entries is not meaningful at ring 2.
- Two new commands were added to read and write ranges of memory of a specified length. The read-buffer command was added to replace the missing read-only alias command. The write-buffer command was then added for symmetry.
- The Debug buffer was reformatted to hold linear addresses, and no Debug command references memory via selectors. Debuggers are expected to understand the relationship between LDT addresses and the linear memory region.

While reviewing the architecture section for portability and 0:32 memory model, the following miscellaneous changes were made.

- The 286 version of PTRACE was scrapped, and will not be supported at all in the 386 system. The main reason for this was its inability to get to the D-bit and B-bit of the segments. This ability is crucial to debugging in the 386 environment. This means that there will be no version translation code in the kernel, and that old debuggers will fail to load on the 386 system.
- The Access rights and Limits for all segment registers was placed into the Debug buffer for debugger performance.
- PTRACE was renamed Debug to prevent confusion between the versions.

Debug Design

This page is intentionally left blank.

Design Overview

All processes are started via an DosExecPgm call. If the option to debug the process is given to DosExecPgm, the process will have a Debug Control Block (DCB) allocated and attached to it, and the FF_TRC force flag will be set on it's initial thread. This DCB is used to time and control the execution of the debuggee process. FF_TRC is used so that the debuggee thread will be directed into the Debug component and will wait there via a DosSleep until instructed to run user code by the debugger.

A Debugger application calls Debug, passing a pointer to a buffer, which is known as the user Debug Buffer (uDB). This buffer is copied into a ring 0 kernel Debug Buffer (kDB) temporary buffer.

The kDB buffer contains the Pid of the debuggee process of interest. The DCB for that debuggee process is located and verified as early as possible in the Debug processing.

The kDB buffer also contains a command number, and the parameters required to fulfil that command. If the command can be performed by simply using the debugger thread to operate on the debuggee process, the command is completed in that way.

However, some commands result in running debuggee code, such as the Go or SStep commands. In this case, the editing is done to the Debuggee's TCB and PTDA by the debugger to set it up to run. The Debuggee is then instructed to run by setting a flag in the DCB, and a wake-up is done to wake it up, by the debugger. The debugger then waits via a DosSleep call until the debuggee has completed running the requested code.

When the debuggee wakes up, it (usually) attempts to run user code. The debuggee process runs just as any other process until another debug event occurs, such as a single step, breakpoint, or process termination. When this event occurs, the debuggee sets the proper fields in the kDB to indicate the appropriate event, wakes up the debugger, and calls DosSleep again. This is known as a notification.

When the debugger wakes up from the event, it then copies the kDB back to the location of the original uDB, and returns control to the debugger application.

Internally, Debug is made up of a number of smaller sub components working together. These sub components handle :

- Initialization
- The Debugger routines
- The Debuggee routines
- Watchpoint and Debug Register management
- kDB management (local variables and shared data packet)
- DCB management (per-debuggee variables)

Some special features are included in the internals of Debug that assist this basic design. Some of these features include :

1. Single Step Simulation

Ring 0 code may not be debugged by the users of the Debug function. To do this, Single Step Simulation of callgates is required.

For single-stepping to be controlled by ring 0, but occur in user space (ring 2 or 3) the TF bit in the eflags must be turned on via an IRET (or POPF, RET) instruction which causes a privilege level transition (PLT). This is because the PLT Iret stack frame contains an image of the eflags, as opposed to a PLT Retf instruction, which does not.

For Debugging user code which does not use a callgate, Single Stepping may progress a single instruction at a time quite easily, because every Single Step causes an exception (exception 1), which automatically causes the PLT Iret stack frame to be placed on the kernel stack, and the TF bit is automatically cleared in the eflags. So, the stack is set up for the next Single Step just as the previous Single Step completes.

However, a callgate to ring 0 causes some drastic action to take place. The stack frame which would eventually be used to return to user space does not contain the eflags image, and the stack frame which would normally be used to single step points to the first instruction within the API call at ring 0. Note that the first instruction of the API call has not yet been executed.

At this point, the ring 0 stack frame is difficult to deal with, but is completely known. The ring 0 stack frame looks like the following upon entrance to the trap01 routine.

```
Stack Base ---> |-----|
                  | user SS:ESP   | retf frame
                  | <N parameters>|
                  | user CSE:IP   |
                  |-----|
                  | EFlags image  | exception 1 frame
                  | Kernel CSE:IP  |
SS:ESP          ---> |-----|
```

To avoid single stepping ring 0 code, the trace bit is turned off in the eflags image and the thread is allowed to continue execution. When the API call is complete, the debuggee thread will attempt to leave the kernel. All debuggee threads have the trace bit in their flags image validated before leaving the kernel, so the trace bit will be turned on in the debuggee's flags image. The last two instructions before returning to user code from the kernel are always POPF, RETF. The POPF will restore the trace bit from the flags image, and a single step trap will occur after executing the RETD.

This gives the user the impression that the API call executed as a single instruction.

2. Command Serialization

Some Debug Commands require serialization. These are the commands which use the debuggee process to do the work, or commands which absolutely require a stable debugging environment to be trusted. At the very least, the following commands must be serialized.

- DBG_C_WriteReg
- DBG_C_WriteCoRegs
- DBG_C_Go
- DBG_C_Term
- DBG_C_SStep

The DBG_C_Stop command is also serialized in some cases.

Internally, the DebRequestSem and DebClearSem procedures provide this serialization, which is accomplished via a simple RAM semaphore existing in the DCB.

The other commands may be executed while the debuggee process is executing user space code, and provide a snapshot mechanism to assist performance.

3. Debuggee Cleanup

A debuggee process is unable to terminate itself, but must be told to terminate either via a terminate command, or a Go or SStep command which occurs following process termination notification. However, there are some special cases which require attention.

The first case is when a process is started for debugging, but no debugger process decides to control the process. To solve this problem, there is a fixed amount of time allowed between starting a debuggee process and connecting a debugger process to that debuggee. If this takes more than a few minutes, it is then assumed that the debugger will never connect to the debuggee, and the debuggee process is terminated.

The second case is when a debugger process dies without using the proper commands to explicitly terminate the debuggee. This could occur if the debugger was unexpectedly terminated via a GP fault during debugging. If a debugger terminates, all of the debuggee processes it had connected to are automatically terminated. A debuggee process is conceptually considered to be a loosely connected resource owned by the debugger, and the debugger is expected to manage this resource.

4. FF_TRC Force Flag

The FF_TRC force flag is used to assist Debug. FF_TRC is used to signal some types of Debug events, when the events may occur asynchronously with respect to the debuggee thread. Events of this type include the Initialization, Freeze, AsyncStop, and Watchpoint events.

If a thread ever attempts to use Exit_Kmode when FF_TRC is set, the thread will enter the DbgEvent procedure. DbgEvent then checks additional flags that were set at the same time that FF_TRC was set, to determine why FF_TRC was set. DbgEvent will then take appropriate action, which may include notifying the debugger of an event.

5. Pending Notifications

Sometimes, Debug is required to return more information than can possibly be returned in a single notification. In this case, the extra notifications become pending. These pending notifications can occur from multiple watchpoint hits or library loads. Watchpoints are particularly interesting, as they can become pending at any time.

There is also a class of events which are not as important as other events, and can be overridden if another, more important event occurs. An example of this type of event is the Async Stop event.

For instance, if an AsyncStop event occurs at the same instant that a watchpoint gets hit, the watchpoint hit may override the AsyncStop event, because the watchpoint hit notification will serve the same purpose as an AsyncStop notification.

6. Watchpoint and Debug Register Management

Watchpoints in Debug are implemented via the 80386 Debug Registers.

Watchpoints can have two scopes, being Global or Local. Debug registers, however, implicitly have the scope controlled by the current linear arena. This means that the Debug Registers need to be context switched to accommodate watchpoints in the local scope. This context switching is accomplished by a call to DbgWPValidate, which will set all the debug registers to whatever values are correct in the current context, regardless of their previous values.

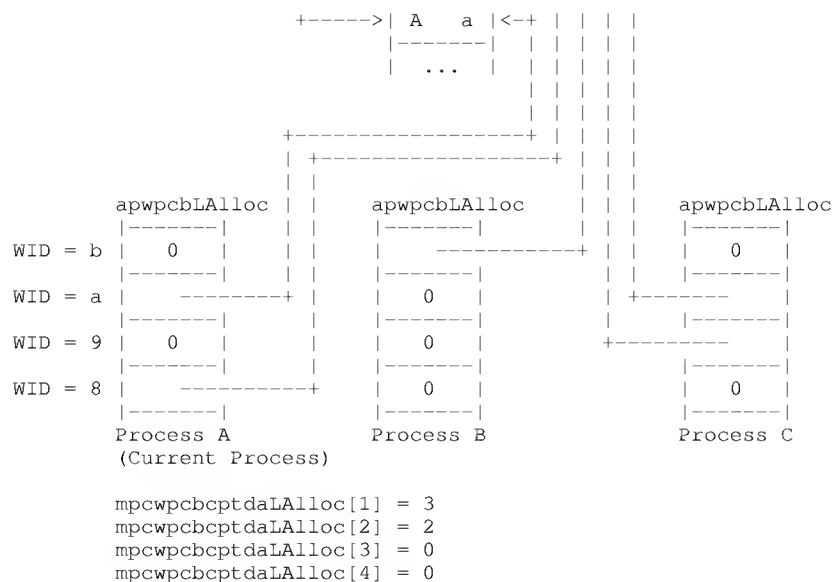
Because the relationship between watchpoints and the debug registers is hidden from the debugger, the watchpoints may be virtualized, using the debug registers as needed.

Each allocated watchpoint has a WPBC (Watchpoint control block) associated with it. These WPBCs each contain all the information required to track the state of the watchpoints, update the debug registers, and notify the debugger. The WPBC also contains identification information, including a reference to the owning debuggee PTDA, and the WID (Watchpoint ID). The WPBCs will be allocated from either a BMP managed object, or perhaps from the kernel heap. They will not be swappable.

Pointers to these WPBCs exist in 3 places. First, there is an array of 4 pointers to WPBCs (one for each debug register) from each PTDA (apwpcbLAlloc), which specifies the local watchpoint allocations. There is another set of 4 pointers to WPBCs from DGroup which specify the global watchpoint allocations (apwpcbGAlloc). The third set of 4 pointers to WPBCs from DGroup is maintained in parallel with the actual debug registers on the CPU, which identifies the correspondence between any particular debug register, and the Watchpoint control block from which its contents were last generated (apwpcbCur).

The pointers from apwpcbCur have one possible additional state besides being NULL (free) or a valid pointer to a WPCB. This state is represented as a -1, and means that control of the debug register has been taken over by either the kernel debugger or the ICE386. In this case, neither this debug register nor any CPU flags in DR6 and DR7 corresponding to them will be modified by the kernel. This enables the kernel debugger to manage its own debug register breakpoints, as needed.

	apwpcbCur	WPCB	apwpcbGAlloc
DR3 3	-----+ +-->	C d <-----+	0 WID = f
DR2 2	-1	C a <-----+	0 WID = e
DR1 1	-----+	C 9 <-----+	+---- WID = d
DR0 0	-----+	B b <-----+	0 WID = c
	+--->	A 8 <---+	(Global)



This diagram shows a possible Watchpoint allocation snapshot. Process A has 2 local watchpoints allocated to it (WID = '8' and 'a'); process B has 1 local watchpoint allocated (WID = 'b'); process C has 2 local watchpoints (WID = '9' and 'a') and one global watchpoint (WID = 'd') allocated to it. The Kernel debugger has reserved debug register 2.

Each WPCB contains (among other things) a pointer to a PTDA, and a WID (Watchpoint ID), which is represented above by a letter and a number in the WPCB. This is used to identify the debuggee process which is using the watchpoint, and exactly which watchpoint it is within that process.

Watchpoint allocation is driven by a single basic rule. There may be an absolute maximum of 4 CPU debug registers in use at any point in time, whether in use by the kernel debugger, a global watchpoint, or a local watchpoint. Allocation of a global watchpoint for a process, or reservation of a debug register for the kernel debugger, results in decreasing the number of possible local allocations for each process in the system (by 1).

Allocation of global watchpoints for processes, and debug registers for the kernel debugger, is assisted by keeping a 4 byte array in DGroup which counts the number of processes with at least a certain number of local watchpoints allocated. The name of this array is `mpcwpcbcbptdaLAlloc`, which is shown above. `mpcwpcbcbptdaLAlloc` stands for a mapping array from a count of Locally Allocated WPCBs to a count of PTDA's (Processes). This gives the debug register allocator the ability to very quickly find the current maximum number of local watchpoints in use across all the processes. Because the kernel debugger must be able to allocate a debug register at interrupt time, it is not possible to scan the PTDA's to find this information.

Allocation of a local watchpoint does not require scanning every debuggee in the system, but does require cross checking that there will be enough debug registers available within that one process context.

A process may allocate a local watchpoint if the following checks pass :

A local watchpoint slot is available in the PTDA, and

The sum of the following is less than 4 :

- The number of reserved debug registers.
- The number of Globally allocated watchpoints.
- The number of Locally allocated watchpoints within the current context.

When a process allocates a local watchpoint, it counts the number of local watchpoints it finished with, (call this `cwpcbLAlloc`) and simply increments `mpcwpcbcbptdaLAlloc[cwpcbLAlloc]`. When the watchpoint is freed again, the same location is decremented.

A process may allocate a global watchpoint, or the kernel debugger may reserve a debug register, only if the following checks pass :

A global watchpoint slot is available, and

The sum of the following is less than 4 :

- The number of reserved debug registers
- The number of Globally allocated watchpoints
- The maximum number of Locally allocated watchpoints in use by each process in the system.

The algorithm for calculating the maximum number of Locally allocated watchpoints in use by each process in the system traverses `mpcwpbcptdaLAlloc` as opposed to scanning all the processes. For example, if no process has more than 2 locally allocated watchpoints, then `mpcwpbcptdaLAlloc[2]` must contain a nonzero, while `mpcwpbcptdaLAlloc[3]` and `mpcwpbcptdaLAlloc[4]` must contain zero.

It may be educational for the reader to examine the above diagram to show that the only possible remaining allocation (without using another PTDA) is a single local allocation by process B.

The validation of the debug registers occurs in stages. First, the Debug registers are disabled by clearing some bits in DR7. Next, `apwpcbCur` is filled in with pointers to the new current WPCBs which are appropriate for the current context. `apwpcbCur` is then traversed (skipping the invalid entries), filling in DR0 through DR3 with the linear addresses from the current WPCBs. Finally, the proper bits are turned on again in DR7, enabling the debug registers.

The routing of a debug register hit at tasktime is done by following the appropriate pointer in `apwpcbCur` to the correct WPCB. From there, the owning PTDA is quickly identified, so that `FF_TRC` may be set on that process so that it may respond to the event. The PTDA pointer in the WPCB is guaranteed to be valid, because each process frees all remaining attached WPCBs before evaporating.

The notification of an interrupt time watchpoint hit is delayed until the earliest possible tasktime convenience. If a watchpoint is hit at interrupt time, the debug register which that watchpoint was using is disabled by turning off `Gn` in DR7, the notification information is stored, and a flag is set to signal a watchpoint hit.

The coding watchpoint IDs (WID) was designed with the following goals in mind.

- A watchpoint ID of zero should be invalid, for possible future use.

- A watchpoint ID should aid in locating the watchpoint control block quickly.

- The values given to a watchpoint ID will not be documented.

- The encryption of watchpoint IDs may change from version to version.

- The values of local watchpoint IDs need not be different for separate processes.

The use of a watchpoint ID to quickly locate a watchpoint control block prompted the encryption of the scope and the index of the watchpoint into the watchpoint. Because an array index is used to locate a `wpccb` of either scope, the index is stored in the least significant 2 bits. The scope is stored in the next bit, being 1 if global, or 0 if a local scope watchpoint.

While this encryption is very useful in locating a watchpoint quickly, taking a simple mask to generate either the scope or index, it suffers from being zero in the case one of the local watchpoints. To counter this, the next bit up must always be a 1, and the remaining bits will be zeros.

In summary, a WID is composed of the following elements.

- Bits 0-1 : Index to an array of 4 pointers to watchpoint control blocks

- Bit 2 : ON if global, OFF if local.

- Bit 3 : Always ON

- Remaining bits must be OFF.

7. DCB management

Each debuggee process in the system, has a DCB (Debug control block) attached to it. Because of this, the DCB can serve some side roles besides just controlling timing of data passing between the debugger and the debuggee. The existence of the pointer to the DCB in the PTDA can be used to tell if a process is currently debugged.

The DCBs will be linked onto a chain. By traversing this chain, the debuggee processes may be enumerated faster than traversing every process in the system. This is the method which will be used to locate debuggee processes from the passed PID.

8. kDB management

The kDB is a data block used for holding variables local to an instance of a call to Debug, and is also used to pass data between the debugger and the debuggee process.

Because the kDB is small enough, (approx 260 bytes), and needs to exist only for the duration of the Debug call, the kDB will be allocated from the debugger thread's kernel stack.

Because the kDB is used to pass data from the debuggee to the debugger, this implies that the debuggee thread must directly modify the stack of the debugger thread. This can pose a maintainability problem if not carefully controlled.

However, this problem can be completely contained by noting that the only time that the debuggee needs to access the kDB is when it is running the `DbgNotify` procedure, which returns data to the debugger. By the time `DbgNotify` is called, the entire notification is known, so the kDB can be accessed by the debuggee in a single pass, without blocking. Prior to accessing the kDB,

the debuggee kernel code will verify that a pointer to the kDB exists in the DCB.

The debugger can further reduce the likelihood of these errors by noting that the pointer to the kDB in the DCB should be nonzero only when the debugger is actually waiting on the debuggee process to finish a command. Naturally, the passed kDB pointer will be a DS-relative pointer rather than SS-relative.

Exported Data Structures Description

Local Data Structures Description

Exported Interfaces

Imported Interfaces

The following procedures and data, or their equivalents, will be required to complete the DosDebug function :

- Global Data :

```
DCB_t    *pdcbbHead      = NULL;
int       DBGcSysDR        = 0;
int       DBGckDR          = 0;
WPCB_t    *DBGapgwpcb[DR_COUNT] = {NULL, NULL, NULL, NULL};
int       DBGaDRowner[DR_COUNT] = {0, 0, 0, 0};
long      DBGIDR7global;    = 0;
ushort_t  DBGmpclwpcbcptda[DR_COUNT] = {0, 0, 0, 0};
long      DBGImaskDR7system = DR7_RESERVED;
long      DBGImaskDR7reserved = DR7_RESERVED;
DCB_t     *DBGpdcbbCur     = NULL;
```

- PTDA Data :

```
DCB_t     *ptda_pdcbb     = NULL;
ulong_t    ptda_fIDbg      = 0;
```

- TCB Data :

```
ulong_t    TCB_fIDbg      = 0;
```

- Tasking (TK) Component

TKDieSoonTask (pptda, Exit_Type, Exit_Code);

TKRun (Block ID);

TKBlock (BlockID, Timeout Length, Interruptable Flag);

TKSuBuff (pchDest, pchSrc, cchLen);

TKFuBuff (pchSrc, pchDest, cchLen);

TKSetForce (FF_TRC, pptda);

TKUserMemCopy (pptdaSrc, pchSrc, pptdaDest, pchDest, cchLen, plfOptions);

plfOptions contains the following bit flags :

Input : make read-only destination pages private

Input : Kernel space addresses for source are valid

Input : Kernel space addresses for destination are valid

Output : In case of a fault, an indication of whether the source or the destination faulted.

ptcb = TKTidToTCB (pptda, Tid);

pptda-> sPid (Process ID)

ptcb-> sTid (Thread ID)

ptcb-> iSlot (Thread Slot Index)

pptda-> ptcbHead

ptcb-> ptcbNext

Indication of Embryo process state while being Exec'ed.

- Kernel Heap Manager Component

pch = malloc (cchSize);

free (pch);

- Dispatcher Component

A call to DbgEvent from CheckForceFlags if FF_TRC is set.

Read/Write Access to all user's registers. (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, CS, DS, ES, FS, GS, SS, EFlags).

An indication of whether the current stack frame for any given thread is :

An Iret frame.

A Callgate frame.

Currently unknown (InDos == FALSE).

For a known callgate stack frame, the number of parameter bytes.

Read and Write access to the FF_TRC force flag for each thread.

- Signals Component

A call to DbgSignal

A bit-vector indication of pending signals (sig_pend)

- Memory Manager Component

VMMapDebugAlias (ppchAlias, hptdaTarget, pchTarget, cchLen, fWriteable);

VMFreeAlias (ppchAlias, hptda);

<Access, Attribute, Limit> = GetDescInfo (pptda, Selector);

A Call to DbgAliasFree (ppchAlias, hptda) when freeing an aliased object.

- Exception Handling Component

Linear address (CR2 contents) of the most recent page fault by this thread, if a Fatal Page Fault.

A Call to DbgWPScan by trap01 routine.

A Call to DbgException by pentry routine for fatal faults.

SetSimGPFault (ptcb);

- DLL Library Initialization Component

hnte handle Generator for LibLoad notifications.

A Call to DBGLibLoadEvent when a libload event is detected.

- NPX Manager Component

NPXReadContext (Slot, Addr, Length, Reserved = 0);

NPXWriteContext (Slot, Addr, Length, Reserved = 0);

An indication of whether a 80286 or 80386 is in use by a thread.

- Loader Component

LDRIsMteValid (hmte);

LDRNumToAddr (ppchAddr, hmte, Object Number);

LDRGetLibName (pchBuffer, hmte);

LDRGetSem ();

LDRClearSem ();

- Block Management Package (BMP) Component

pch = BMPGetTempBuffer ();

BMPFreeTempBuffer (pch);

- Scheduler Component

int SCHGetState (ptcb)

int SCHGetPriority (ptcb)

A compile-time list of possible states from SCHGetState.

- Required Memory Conversions

pch = Linearize (Sel:Off32)

DS-relative Address = SStoDS (Stack-relative address); Both C-callable and MASM-callable versions will be required.

- Standard C Macros

FIELDOFFSET(type, field)

InternalError(message)

The former names of these functions are :

- pptda-> IfDbg was PtraceFlags
- pptda-> pdcb was PtraceEnabled, or PTCBOff
- pptda-> IfDbg was TCBPtraceFlags
- pptda-> lselDbgCS was TCBTraceCSIP.segmt
- pptda-> loffDbgEIP was TCBTraceCSIP.offst
- TKDieSoonTask was DieSoonPTDA

- TKRun was ProcRun
- TKBlock was ProcBlock
- TKSuBuff was SuBuff
- TKFuBuff was FuBuff
- TKSetForce was Set_Force or QuietSetForce
- TKTidToTCB was TidToTCB
- sTid was TCBOrdinal
- iSlot was TCBNumber
- pptda-> ptcbHead was ActiveList
- ptcb-> ptcbNext was TCBNext
- DbgEvent was TRCNotify
- VMMapDebugAlias was VMMapPtraceAlias
- VMGetDescInfo was VMGetDescInfo
- LDRIsMteValid was CheckModHandle
- LDRNumToAddr was num_to_sel
- LDRGetLibName was GetLibName
- LDRGetSem was GetLdrSrm
- LDRClearSem was ClearLdrSem
- BMPGetTempBuffer was GetTempBuffer
- BMPFreeTempBuffer was FreeTempBuffer
- SCHGetState was GetState
- SCHGetPriority was GetPriority

Previously Unused Existing Interfaces

- GetDescInfo
- SetSimGPFault
- FIELDOFFSET

New Imported Interfaces :

- TKUserMemCopy
- Indication of Embryo process state
- Kernel heap malloc
- Kernel heap free
- A call to DbgWPDelayed if (resched\$dointdisp) is set
- Linearize (convert 16:32 to 0:32)
- SStoDS (convert SS-relative pointer to DS-relative)
- VMMapDebugAlias
- VMFreeAlias
- Saved CR2 contents after Page Fault
- A Call to DbgWPScan by trap01 routine
- A Call to DbgException by pentry routine
- Ability to temporarily edit the trap01 vector
- An mte handle Generator for LibInit notifications
- NPXReadContext
- NPXWriteContext
- An indication of whether a 80286 or 80386 is in use by a thread

Design Constraints

Debug must have a minimal affect on processes which are not being debugged. Debug should not be a drain on system resources when not in use, but should be quick when in use. Currently, there are no fixed performance goals for Debug.

Debug must be as portable as possible. Therefore, almost all of the code will be written in C.

Debug Implementation

This page is intentionally left blank.

Implementation

See procedure headers for pseudocode.

Implementation Review

Comments from DosDebug Design Overview Meeting Attendees

Presenter

rickdew

Reviewers

briansm

alex

danno

donh

ianb

laled

larrys

jameelh

petes

vich

Missing reviewers

kevinro

raype

Questions by Reviewers

(vich) Should we make DCBs swappable?

Is it possible?

rickdew

Yes. In Ptrace, the PTCB (the old analogue to the DCB) was swappable.

However, unless a close review is done allowing for preemption any time the DCB is referenced, this could pose a problem, resulting in rare bugs which could potentially crash the system.

For instance, when referencing the DCB, the debuggee process may evaporate, invalidating pdcB->pptdaDbgee.

This may not be a significant problem, as only the first reference to the DCB following a block may block. So, when referencing the DCB a few times in a row, we will block only on the first reference.

Should it be done ?

vich

yes. All objects should be swappable by default, and nonswappable only when absolutely necessary.

rickdew

yes, with caution. I feel that there is only a minimal gain by doing this.

The tradeoff may not be an interesting one, as DCBs are only 24 bytes long, are rare in general, are heavily accessed when in use, and swapping them can cause hidden bugs.

Action

Under debate, but the final action will be to either allocate DCB's from the nonswappable heap, or to allocate DCB's from the swappable heap, with a close review of the code.

(petes) Does DosDebug handle having interrupts disabled for IOPL Debuggees?

rickdew : No.

DosPtrace will also fail in this case, and it is not possible to support.

Let's take the scenario where a debuggee hits a breakpoint (int3) when it has interrupts disabled.

The only options available include either stopping the debuggee to notify the debugger, or restarting the instruction.

Well, restarting the instruction is not possible, as another int3 will occur immediately, creating an infinite loop.

So, the ONLY option is stopping the debuggee in this case.

While it is possible to restart a debug register hit, this defeats the purpose of the debug registers. In addition, this is very difficult to support.

If a debug register is hit while the debuggee has interrupts disabled, the debuggee will be stopped.

I will Document that Debug is will have an effect on a debuggee which runs with interrupts disabled. The debugger should check the flags word automatically when it reads the registers, and issue a warning if an event occurs with interrupts enabled.

This is not a major problem, as we don't do a wonderful job of supporting cli/sti in a paged system, anyway, and the concept is virtually useless in a multiprocessor system.

(petes) Does DosDebug handle taking a debug register hit in a process while it has interrupts disabled?

Problem

There are two interesting cases here.

1. A watchpoint hit, taken while in ring 0, when interrupts were disabled. This includes both interrupt time and task time areas of the kernel, device drivers, ring 0 DLLs, and FSD's.
2. A global watchpoint hit, taken while running user code in a process where interrupts are disabled, and where that process is not the same process which owns the watchpoint. The process hitting the watchpoint should not be affected in this case, as it is not actually being debugged.

Solution

For both of the above cases, interrupts must remain disabled at all times.

This implies :

- The gate in the IDT for the trap01 handler must be an interrupt gate.

This is already true.

- DynamicTrace must complete its int1 processing without enabling interrupts.

This is already true. Static RAS is expected to function at interrupt time, and RASDynTrcTrap1 calls static RAS to do most of the work.

- SetSimSStep must complete its processing without enabling interrupts.

This is already true.

SetSimSStep has no need for enabling interrupts, as it just does a kernel stack munge. SetSimSStep is written in C.

- DbgWPScan must complete its processing without enabling interrupts.

This is already true.

DbgWPScan is written in C, and does not access any swappable data. It does not enable interrupts.

- For case (1) above, (a hit at ring 0) the hit is delayed until Exit_Kmode, where interrupts are enabled again, so ring 0 is not a problem.
- (BUG) For case (2) above (a hit while not at ring 0), the trap01 handler was supposed to call Enter_KMode and Exit_KMode to handle the hit. However, to support interrupt time hits, the trap01 handler will need to do the following.

```

If (NONE OF (B0, B1, B2, B3, BT, BS, BD) IS SET IN DR6) {

    set BS in DR6 to 'simulate' a single step.

    If the user issues a plain INT 1 opcode,
    we will pretend that it was a single step,
    to achieve compatibility with the previous
    OS/2 versions. (an int1 in a debuggee causes
    a single step notification)

}

handle RASDynTrcTrap1, clearing BS if applicable

handle SetSimSStep, clearing BS if applicable

Save state of FF_TRC in a local variable

if ( (DbgWPScan() OR BS) AND (not ring 0)) {

    if (pptdaCur-> flDbg : DBGP_WPHALFHIT) {

        Restore FF_TRC to it's previous value.
        If FF_TRC was previously set, it will
        remain set, preventing us missing a
        Debug event. If previously clear,
        and DbgWPFinishHit sets it, then
        we will call Enter_Kmode soon.

        The key point is to keep FF_TRC clear
        in the case where the current process
        is not the debuggee which owns the
        watchpoint.

        Tell tasking (using RAL?) where the
        instruction pointer is, so that
        DbgWPFinishHit works.

        DbgWPFinishHit ();

        Erase pointer to the instruction pointer.

    }

    if ((FF_TRC is set) OR BS)) {

        if (DebugEnabled) {

            Enter_Kmode ();

            if (BS) {
                Clear BS;
                DbgException (SStep exception);
            }

            Exit_Kmode ();

        } else {

            Do nothing.

            BS was set in the context of
            a non-debugged process.

            There is no (known) way that
            FF_TRC can be set here.

            MISCSTRICT :

```

```

        if (FF_TRC set)
            InternalError ()

        We will jump to pentry soon.
    }
} else {

    Do nothing.

    A global watchpoint was hit at some
    ring other than 0, outside the context
    of the process which owns the
    watchpoint, so we keep it running.

    Interrupts were disabled throughout
    the processing of this exception, so
    that the current process will not be
    affected.
}
} else {

    Do nothing, because one of the following has
    happened :

    1) BS was set on entry, but was cleared by
       RASDynTrcTrap1 or SetSimSStep.

    2) No debug register was hit which also
       belongs to Debug. (DbgWPScan == FALSE)
       Could have been a hit for the kernel debugger.

    3) A watchpoint was hit at ring 0, so we must
       delay the hit until Exit_Kmode will be called
       at a more convenient time.
}

if (ANY OF (B0, B1, B2, B3, BS, BT, BD) IS SET IN DR6) {
    jump to pentry;
} else {
    iret;
}

```

(petes) What restrictions or assumptions exist which will be invalidated on a multiprocessor system?

rickdew

This should be determined by a complete review of the code. However, it appears that the only areas which could possibly be affected are in the debugger and the communication areas. That is, the routines which run entirely in the debuggee context are not at risk, as they only access the current context.

Known Assumptions

- non-preemptive kernel in some areas
- Debuggee process may not terminate while the debugger is operating on it.
- All threads except the current thread have valid stack frames.
- Debugger kernel stack is valid while the debuggee thread is accessing it.

Possible Correction of stack frame problem.

- Issue a 'request stack frame' call. This would mark the debuggee thread to stop in the kernel at the earliest convenient point, handle the request, and continue.

Changes required for a multiprocessor environment.

- API changes, TBD.
- Move the kDB to the kernel heap, off of the debugger kernel stack, and add a reference count to it.
- May require changes to how we read/write the registers.
- Other unknown changes, depending on the general architecture we choose support.

(vich) What is the precise scope of watchpoints?

Any watchpoint may be hit from user mode, kernel mode, or Interrupt time.

Local Watchpoints are effective in one (debuggee) context only

Global Watchpoints are effective in all contexts at once, on shared address regions only

Assumptions

For any particular address in the shared flat region of any protected mode process, one of the following is true in the context of each process in the system :

1. The address is valid, and points to a shared physical memory page.

In this case, the debug register must point to the given linear address.

2. The address is valid, but points to a private physical memory page. This can only happen if the page was written to via a Debug write-memory command, which made the physical page private. This will only occur on read-only pages.

In this case, the debug register may either point to the given linear address, or be disabled.

To determine which of these to do, we examine what the user would like to see. Most of all, the user wants consistency, so this scenario should function just like scenario (a), if possible. By leaving the watchpoint set, we will hide the privatization of the page, and maintain this consistency.

Also, this would slow down the debug register context time by about 12-20 times in the normal case. By examining the other cases, it is sufficient to always leave the watchpoint enabled. This is about as easy as possible, as it involves just doing a pointer dereference to get the debug register value, and stuffing it in, no questions asked.

Just asking the question as to whether the underlying physical memory is private memory belonging to some process other than the debuggee would take about 3-5 times longer than simply stuffing the debug register, per register. Even if the test was extremely simple, the test is guaranteed to be overhead, and will need to occur 4 times in a row. (overhead = 3-5 times longer * 4 registers = 12-20 times the context switch time)

3. The address is invalid in the context of a protected mode process, because that process has not requested access to that memory.

The debug register may remain enabled, as the underlying memory is invalid, and will take a page fault before taking a debug register hit.

4. The address is invalid, because it is in the context of a V86 mode process.

This case is the same as above. The debug register may remain enabled, but will not be hit because it is invalid.

(vich) Should we support global watchpoints to private regions of the address space?

rickdew : No.

First, this significantly decreases the context switch time for debug registers, to turn them on only in the correct context. See (b) in the above discussion.

Secondly, it does not make a lot of sense. If some memory lies in the private region of the address space, it can not be modified out of context. If it is, this would imply that the memory manager is broken.

However, the memory can be read and executed out of context. The only question that can be answered by this is : "What other processes are running in the system which are using this same executable image?" This is only very rarely interesting, and would actually be more confusing than informative.

(vich) Should the debugger see some special code that says that the watchpoint was hit at interrupt time? Perhaps we can return a bunch of zeros in the notification?

rickdew

Oh, my. I was supposed to be returning Zeros in the interrupt time watchpoint hit notification, and I'm not. Oops!!

I will make pcode changes to DbgWPDelayed.

(petes) What is the effect on Debug for ring 0 DLLs ?

rickdew

I don't know enough about ring 0 DLLs to answer the question completely.

It might not even be wise to allow debugging on these particular regions of code at all.

What does the ring 0 stack frame image of a ring 0 DLL look like?

Given that the debuggee has stopped somewhere inside a ring 0 DLL, what is the register set? Is it the ring 0 register set, or the ring 3 register set? How will we get to the ring 3 register set, if the ring 0 DLL did not save the image in a consistent manner?

Do we allow access to the memory of a ring 0 DLL?

(larrys) What interface should the loader call to register an event? It can't be DbgLibLoad, because that is a private interface.

rickdew: I will add one.

The name will probably be DbgLibLoadEvent(), given no parameters. This will not absolve the LibInit code from its duties, however.

Debug Appendix (included for each review)

This page is intentionally left blank.

Debugging features

DBGImaskDR7system - mask of systemwide valid DR7 bits (includes bits in use by both DosDebug global watchpoints and the kernel debugger, as well as processor reserved bits).

DBGImaskDR7reserved - mask of bits not used by DosDebug (includes bits in use by the kernel debugger and processor reserved bits).

DBGpdcbaCur - DCB for current local watchpoints

DBGP - DosDebug Per-Process Equates

DBGT - DosDebug Per-Thread Equates

DBGS - DosDebug SeverConnection Equates

DbgKillDebuggees - Kill All Debuggee processes of current process

DbgEnable - Enable DosDebug on a Process

DbgDisable - Disable DosDebug on a Process

VR32Debug - Verification Routine for the DosDebug API function

DbgTrap - Notify debugger of a single step or breakpoint trap

DbgException - Notify debugger of an exception

DbgNewProc - Notify the debugger of a new process

DbgAliasFree - Notify the debugger that an alias needs to be freed

DbgThreadTerm - Notify the debugger that a thread is terminating

DbgProcTerm - Notify the debugger that the process is terminating

DBGEntryEvent - handle a debug entry event

DbgEvent - Handle an asynchronous debug event via TK_FF_TRC

DbgDLLEvent - Mark thread to pick up DLL notification

DbgValidateTF - Validate this thread's user TF bit

DbgWPHit - Route a watchpoint hit to a debuggee

DbgDRAlloc - Request to lock a debug register

DbgDRFree - Unlock a locked Debug Register

DBGWPAIlocVDM - allocate a watchpoint for a VDM

DBGWPFfreeVDM - free a watchpoint for a VDM

DBGDR6QueryVDM - query/update DR6 for a VDM process

*****EP DbgWPScan** - Scan watchpoints

DbgWPScan scans the CPU debug registers to see if a watchpoint was hit.

DbgWPScan may be called at interrupt time.

If DbgWPScan detects a debug register hit that is intended for DosDebug, the Bn bit in the DR6 image and the Gn and Ln bits in DR7 will be cleared, and DbgWPHit will be called to route the hit to the debuggee.

Debug registers which are hit but are not intended for DosDebug will remain hit.

Debug registers which are hit but not armed are ignored - the Bn bit is cleared but no further action is taken.

```
Entry:  Interrupts off, one or more debug registers hit.
        (EBX) = DR6 image
        (ESI) = DR7 image
        if (faulting address is ring 0)
            (EDX) = 0
        else
            (EDX) = faulting linear address
        DS,ES FLAT

Exit:   if (no more DR6 events)
        Carry Set
    else
        Carry Clear
        (EBX) = updated DR6 image
        DR7 Gn and Ln bits clear for each DosDebug register hit.
        (If all DR7 Gn and Ln bits clear, GE and LE also clear).
```

Uses: EAX, EBX, ECX, EDX, EDI, ESI, DR7

The code is straight line for the most common case: one armed debug register hit owned by DosDebug.

Pseudocode: (High Level)

```
EnterKmode = FALSE

for (each hit debug register) {

    if (not armed) {
        clear Bn in DR6 image
        continue
    }

    if (owner != DosDebug)
        continue

    rc = DbgWPHit(debug register index, faulting address)
```

```

    if (rc.EnterKmode)
        EnterKmode = TRUE

    if (rc.Disarm)
        clear Gn, Ln in DR7
}

if (all Gn and Ln clear in DR7)
    clear GE and LE in DR7

if (EnterKmode || more events in DR6 image)
    clear carry
else
    set carry

```

***EP_DBGDRDisableVDM - disable VDM watchpoints

This procedure is called to disable a set of VDM watchpoints. This procedure is resident because it is called in the middle of context switches. In addition to disabling the current VDM watchpoints, it also reenables any DosDebug global watchpoints.

```

ENTRY    ((SS:ESP)+4) = VDM watchpoint control block
EXIT     Interrupts enabled
USES     DR0-DR7, EAX, ECX, EDX, Flags

```

***EP_DBGDREnableVDM - enable VDM watchpoints

This procedure is called to enable a set of VDM watchpoints. This procedure is resident because it is called in the middle of context switches. In addition to enabling VDM watchpoints, it disables any global DosDebug watchpoints.

```

ENTRY    ((SS:ESP)+4) = VDM watchpoint control block
EXIT     Interrupts enabled
USES     DR0-DR7, EAX, ECX, Flags

```

DCB - Debug Control Block structure

dbgvdm_t - VDM watchpoint data block (DPMI)

WPCB - Watchpoint Control Block structure

DBGBugBug - Global Variable for debugging

DosDebug DeBug Flags (bit settings) are defined in DBGAPI.H **vmOffGSR** - Is MODIFIED by Dbgenable, Dbgdisable and Dbgprocterm

pdcbHead - head of list of DCBs

pSemListHead

hdbgTextLock - lock handle for DEBUGLOCK_TEXT

DBGcSysDR - count of systemwide debug register allocations (includes DosDebug global watchpoints and kernel debugger debug register allocations)

DBGckDR - count of kernel debugger debug register allocations

DBGagwpcb - array of global watchpoint control block pointers

DBGaDRowner - array of debug register owners

DBGIDR7global - DosDebug global watchpoint DR7 bit settings

DBGmpclwpcbcptda - map of a count of local watchpoints to a count of of process with that number of local watchpoints.

DBGDR_ - DosDebug Debug Register Constants

DBGTIMEOUT - Maximum time before debugger connect

DBGDCBS_ - Debug DCB State bits

WPCBS_ - Watchpoint Control Block States

DBGWP_ - Watchpoint Scope and Type masks

DBGWID_ - Watchpoint ID macros

DBG_RESERVED_FLAGS

DBGID_ - DosDebug Block ID macros

vmOffGSR - Is MODIFIED by Dbgenable, Dbgdisable and Dbgprocterm

DBG_V_ - Debug Notification Validity Flags

dbgAssert - cause internal error if assertion fails

DbgLinearToSel - Perform the DBG_C_LinToSel command, this subroutine was added by DCR 1320

DbgSelToLinear - Perform the DBG_C_SelToLin command, this subroutine was added by DCR 1320

DbgRegisterSemList - Register a semaphore list with DOSDEBUG

DbgRequestSem - Enter Debugger Command Critical Section

DbgClearSem - Exit Debugger Command Critical Section

DbgNull - Perform the DBG_C_Null command

dbgReadWrite - Read/write debuggee memory through an alias

DbgRWMem - Read/write debuggee memory through an alias

DbgReadWord - Perform the DBG_C_ReadMem command

DbgWriteWord - Perform the DBG_C_WriteMem command

DbgReadBuf - Perform the DBG_C_ReadBuf command

DbgWriteBuf - Perform the DBG_C_WriteBuf command

DbgReadReg - Read Debuggee CPU Registers (DBG_C_ReadReg)

dbgVerifySel - Verify a new debuggee selector

DbgWriteReg - Write Debuggee CPU Registers (DBG_C_WriteReg)

DbgGo - Perform the DBG_C_Go command

DbgTerm - Perform the DBG_C_Term command

DbgSStep - Perform the DBG_C_SStep and DBG_C_RStep commands

DbgStop - Perform the DBG_C_Stop command

DbgFreezeThaw - Perform the DBG_C_Freeze and DBG_C_Resume commands

DbgNumToAddr - Perform the DBG_C_NumToAddr command

DbgCoRegs - Perform the DBG_C_ReadCoRegs and DBG_C_WriteCoRegs commands

DbgThrdStat - Perform the DBG_C_ThrdStat command

DbgAlias - Perform the DBG_C_***Alias commands

DbgConnect - Perform the DBG_C_Connect command

DbgSetWatch - Perform the DBG_C_SetWatch command

DbgClearWatch - Perform the DBG_C_ClearWatch command

DbgContinue - Perform the DBG_C_CONTINUE command

DbgAddrToObject - Perform the DBG_C_AddrToObject command

dbgXchgOpcode - exchange opcode, single-step, and go

DbgLinearToSel - Perform the DBG_C_LinToSel command, [this subroutine was added by DCR 1320](#)

DbgSelToLinear - Perform the DBG_C_SelToLin command, [this subroutine was added by DCR 1320](#)

DbgRegisterSemList - Register a semaphore list with DOSDEBUG

DbgCall - Let Debuggee Process Execute

DbgkDBRead - Read the kDB from the given uDB

DbgkDBWrite - Write the kDB to its corresponding uDB

DbgLinkDCB - Link a new DCB into the DCB list

DbgGetDCB - Locate and Gain debugger access to a DCB

DbgReleaseDCB - Release access to a DCB

DBG32ApiDebug - Worker Routine for DosDebug

dbgUserCSEIP - return the user's linearized CS:EIP

dbgFinishXchgOp - finish exchange opcode command

DbgThreadCreate - Notify the debugger of a new thread

DbgLibLoad - Directly notify the debugger of a module load

DbgLibFree - Directly notify the debugger of a library free

DbgWatchpoint - Directly notify the debugger of a watchpoint hit

DbgHandler - Handle upper level debuggee communications

DbgNotify - Directly notify the debugger of an event

DbgBlock - Block the current debuggee thread until told to execute

DbgEnterCrit - Enter DbgHandler critical section

DbgExitCrit - Exit DbgHandler critical section

DbgProcessSemList - Process the semaphore lists

DbgSeverConnection - Sever the debugger/debuggee connection.

DbgWPAlloc - Allocate a Watchpoint

dbgGlobalWPFree - Free a global watchpoint

dbgLocalWPFree

DbgWPFree - Free a Watchpoint

DbgWPSet - Set a Watchpoint to an address

*****LP DbgValidateDR7** - validate DR7

Updates DR7 with the current global and local watchpoint settings

```
Clear nonreserved bits in DR7
DR7 |= Global watchpoint bits
if (current process has local watchpoints) {
    DR7 |= Local watchpoint bits
}
```

***LP DbgDR7LeGe - Turn off LE or GE bits for the 386 chip A001

```
Clear nonreserved bits in DR7
DR7 |= Global watchpoint bits
if (current process has local watchpoints) {
    DR7 |= Local watchpoint bits
}
```

[illegible]

*****LP DbgSetDRIaddr** - set a linear address for a debug register

```
DR(dsd_index) = dsd_laddr
```

Glossary

AsyncStop	A signal sent to the debuggee threads to indicate that it should interrupt user-level processing to return to the debugger.
Breakpoint	A one-byte int 3 opcode placed in the code of the debuggee. When executed, this int 3 opcode results in returning control to the debugger.
CallGate	A protected mode mechanism for calling an interior ring (ring 0) from an exterior ring (ring 3). This mechanism does not save an image of the flags on the stack.
Cmd	A field in the Debug buffer. When calling Debug, this is a command number. When Debug returns, this is a notification number. This field describes meaning of the information contained in the rest of the Debug buffer.
CodeView/P	A widely used source-level symbolic debugger produced by Microsoft, and the chief client application for Debug.
Command	A message sent to Debug, describing an action for Debug to perform.
Coprocessor	A processor other than the main CPU, such as the 80287 or 80387 NPX coprocessors.
Debug Registers	A set of registers of the 80386 CPU which provide a data breakpoint capability.
Debuggee	The process being debugged, or controlled, by Debug.
DLL	A Dynamic Link Library, created to provide a functional enhancement for programs. A DLL may be used by several executables at once.

DMA	Direct Memory Access device. This sort of device does not use the CPU for I/O. It uses memory shared at the hardware level for a performance increase.
CWait	An OS/2 function which allows one (parent) process to wait for another (child) process to terminate.
Debug	A function which allows one process to control and examine the execution of another process. This function replaces the PTRACE function on previous versions of OS/2.
Dynamic Trace	A function which allows RAS tracing of Executable programs and Dynamic Link Libraries. This function is not directly related to PTRACE or Debug.
DosExecPgm	A function for initiating the execution a process.
DosExit	A function for terminating a thread or process.
DosExitList	A function for defining and executing routines which are run just as a process is terminating.
DosLoadModule	A function used by a process to attach to a set of DLLs.
memory region	A function for managing sparse memory objects.
PTRACE	Process-Trace is a function which allows one process to control and examine the execution of another process. This function is replaced by the DosDebug function on this and future versions of OS/2.
DosQueryProcStat	A function for querying the status of processes running in the system, available in version 1.2 and later.
Dynamic Trace Tracepoint	A location in a program or DLL which DosDynaicTrace is examining. This is (usually) marked by an int 3 opcode, which invokes a PCODE interpreter to save away a small amount of RAS information. A Tracepoint does not invoke a debugger.
EFlags Register	A register on the 80386 processor which contains status flags. Some of these flags are useful for debugging.
Debug Event	An specific action (such as a breakpoint, or thread termination) that may possibly result in a notification to the debugger.
Exception	A programmed fault, trap, or abort detected by the 80386 processor.
fsave / frestore instructions	Instructions which save (restore) the NPX context to (from) a specified address. These instructions are executed by the NPX processor, and have an extremely machine-dependent format.
Interrupt Time	The section of time when the OS/2 kernel or device driver is servicing an interrupt. During this time, the OS/2 memory manager is assumed to be in an unstable state, so that portions of memory may not be accessible. This is the exact opposite of Task Time.
IPC	Inter Process Communication.
Kernel Debugger	

A component linked into a special version of the system that allows debugging ring 0 and real mode code.

Linearized Instruction Pointer

The linear equivalent (0:32) of the CS and EIP registers. If CS is a GDT selector, it is simply the contents of the EIP register. However, if the CS selector is an LDT selector, the linearized instruction pointer is a combination of the CS and EIP registers.

MTE Handle

Module Table Entry Handle. This number uniquely identifies a module in the system, whether it is an executable or a DLL module.

Debug Notification

The act of returning information to the debugger from the debuggee.

NPX

Numerics Processor Extension. This is a coprocessor used to speed the execution of complex mathematics operations.

Pid

A Process Identifier, which uniquely identifies a process.

PTDA

A Per-Task Data Area. The OS/2 kernel maintains exactly one of these structures per process.

RAS

Reliability, Availability, and Serviceability. On OS/2, RAS consists mainly of the ability to log system execution into a circular buffer of a fixed size, to help diagnose system failure.

RF Bit in EFlags Register

When set, this bit causes any debug fault to be ignored during the next instruction, but not debug traps. This is used for restarting an instruction after a fault when there is an instruction execution watchpoint on that instruction. Normally, debuggers should simply never modify this flag.

Segment Access Rights

Byte 6 of a 386 descriptor.

Segment Attributes

Byte 7 of a 386 descriptor. Debug automatically clears bits 0-4 in every returned Segment Attribute image.

Segment Limits

In most 386 descriptors, the segment limit is the segment size - 1, or the offset of the last accessible byte in that segment. This varies with expand-down segments, and means other things in segments which are neither code nor data.

Single Step

The hardware-controlled execution of code a single instruction at a time.

SStep

A shorthand term for Single Step.

Task Time

That time when OS/2 is not servicing an interrupt. At this time, internal data structures are in a known state. This is the exact opposite of Interrupt Time.

TCB

Thread Control Block. This is a per-thread structure managed by the tasking component.

TF Bit in EFlags Register

When on, this bit tells the processor to indicate exception number 1 after execution of the next instruction. This is used to implement the Single Step operation on the 80386.

Tid

Thread Identifier. This is a number which is used to identify threads within a process.

uDB	user Debug Buffer. This buffer contains information that is passed to and from Debug from a debugger application.
VDM	Virtual Dos Mode. This is similar to DOS, but executes in the v86 mode of the 80386 processor.
Watchpoint	An entity maintained by Debug which describes a condition which when met by the debuggee, causes a notification to be returned to the debugger. See the description of the SetWatch command for a list of the supported watchpoint conditions.
Watchpoint ID Number	A number which identifies a watchpoint within a process.
Xenix	A widely used multiuser operating system. PTRACE was originally modeled after a function in this operating system of the same name.

Size and Performance Considerations

Debug is not used often. One of the main goals of the Debug implementation is to cause as little system overhead as possible when not in use, but still be responsive when in use.

There are no exact external size or performance requirements for Debug.

Implementation Estimates

PTRACE LOC ESTIMATES:

	LOCs	Effort	ELOCs	MM
Externals				
* Add Read/Write 386 Registers Commands	80	0.5	40	0.16
* Add Get/Set Coprocessor Registers Cmds	20	1.0	20	0.08
* Add Read Memory Buffer Command	30	1.5	45	0.18
* Replace NumToSel with NumToAddr	10	1.0	10	0.04
* Add Limit and Access Info to PTB	30	1.0	30	0.12
* Add Exception Control Commands	20	0.5	10	0.04
* Add New Exception Notifications	80	0.5	40	0.16
* Add Watchpoint Set/Clear Commands	70	1.0	70	0.28
* Add Watchpoint Notifications	30	1.0	30	0.12
* Add Connect-To Debuggee Command	20	1.0	20	0.08
* Modify Read/Write Word Routines	30	1.0	30	0.12
* Modify Local Alias Interface	20	1.0	20	0.08
Internals				
* Add PTCBActiveList handling to PTCB management routines.	80	1.0	80	0.32
* Add code to handle the new PTrace Buffer format, in read/write kPTB	30	0.5	15	0.06
* Modify uPTB references	80	0.5	40	0.16
* Add table entries to route new commands	30	0.5	15	0.06
* Add Debug Register Allocation, with ICE386, Kernel Debugger Allocation, Local and Global allocation	100	1.0	100	0.40
* Add Debug Register Details, with real mode, context-switch, memory invalidation.	100	1.5	150	0.60
* Add Debug Register Hit detection and routing.	60	1.5	90	0.36
* Modify TRCNotify to deal with new notifications.	30	0.5	15	0.06

Total 950 870 3.48

This gives an estimated 3.5 Man-Months to Code-Complete.

Approximately 2 weeks worth of effort has already been expended in initial design and procedure header work covered by these estimates.

Final Estimation

LOC = 950
ELOC = 870
Man-Months = 3.5
Work Done = 0.3
Today = 06/27/88
Code Complete = 10/10/88

386 PTrace Design Milestone Estimates

Previous estimates on following dates.
6/28

objectives and functionality selection complete
07/05

API interfaces for commands and notifications complete,
architecture review released.

07/11
all procedures and global data outlined
architecture review complete.

07/18
exported procedure headers complete
07/25

all procedure headers complete, all global data defined
design review released.

08/01
pseudocode for all exported procedures complete
design review complete.

08/08
pseudocode for all exported and most internal procedures complete
08/15

all pseudocode complete
implementation review released.

08/22
pseudo-test unit test plan complete
08/29

pseudo-test complete, pseudocode review complete
implementation review complete.

09/05
external procedure pseudocode-to-code translation complete
09/12

internal procedure pseudocode-to-code translation complete
09/19

unit and function test plan complete, unit test started
09/26

unit testing done, ready for integration
code review released.

10/03
function testing done, code review complete, code complete.
10/10

Tracepoint Summary

This page is intentionally left blank.

Hardware interrupts

***DT Hardware Interrupt - Pre-Invocation Trace

```
TRACE    MINOR=0x0001,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=0 (Timer) "

TRACE    MINOR=0x0002,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1 (Keyboard) "

TRACE    MINOR=0x0003,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=2 (NMI) "

TRACE    MINOR=0x0004,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=3 (Serial Port 2) "

TRACE    MINOR=0x0005,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=4 (Serial Port 1) "

TRACE    MINOR=0x0006,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=5 (Parallel Port 2) "

TRACE    MINOR=0x0007,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=6 (Diskette Controller) "

TRACE    MINOR=0x0008,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=7 (Parallel Port 1) "

TRACE    MINOR=0x0009,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=8 (Realtime Clock) "

TRACE    MINOR=0x000A,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=9"

TRACE    MINOR=0x000B,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=A"

TRACE    MINOR=0x000C,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=B"

TRACE    MINOR=0x000D,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=C"

TRACE    MINOR=0x000E,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=D (Coprocessor) "

TRACE    MINOR=0x000F,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=E (Fixed Disk Controller) "

TRACE    MINOR=0x0010,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=F"

TRACE    MINOR=0x0011,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=10"
```

```

TRACE    MINOR=0x0012,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=11"

TRACE    MINOR=0x0013,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=12"

TRACE    MINOR=0x0014,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=13"

TRACE    MINOR=0x0015,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=14"

TRACE    MINOR=0x0016,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=15"

TRACE    MINOR=0x0017,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=16"

TRACE    MINOR=0x0018,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=17"

TRACE    MINOR=0x0019,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=18"

TRACE    MINOR=0x001A,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=19"

TRACE    MINOR=0x001B,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1A"

TRACE    MINOR=0x001C,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1B"

TRACE    MINOR=0x001D,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1C"

TRACE    MINOR=0x001E,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1D"

TRACE    MINOR=0x001F,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1E"

TRACE    MINOR=0x0020,TP=@STATIC,TYPE=(PRE),
DESC="(OS) Hardware Interrupt Pre-Invocation",
FMT ="Interrupt Level=1F"

```

***DT Hardware Interrupt - Post-Invocation Trace

```

TRACE    MINOR=0x0081,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=0 (Timer) "

TRACE    MINOR=0x0082,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1 (Keyboard) "

TRACE    MINOR=0x0083,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=2 (NMI) "

TRACE    MINOR=0x0084,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=3 (Serial Port 2) "

```

TRACE MINOR=0x0085, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=4 (Serial Port 1)"

TRACE MINOR=0x0086, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=5 (Parallel Port 2)"

TRACE MINOR=0x0087, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=6 (Diskette Controller)"

TRACE MINOR=0x0088, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=7 (Parallel Port 1)"

TRACE MINOR=0x0089, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=8 (Realtime Clock)"

TRACE MINOR=0x008A, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=9"

TRACE MINOR=0x008B, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=A"

TRACE MINOR=0x008C, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=B"

TRACE MINOR=0x008D, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=C"

TRACE MINOR=0x008E, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=D (Coprocessor)"

TRACE MINOR=0x008F, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=E (Fixed Disk Controller)"

TRACE MINOR=0x0090, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=F"

TRACE MINOR=0x0091, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=10"

TRACE MINOR=0x0092, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=11"

TRACE MINOR=0x0093, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=12"

TRACE MINOR=0x0094, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=13"

TRACE MINOR=0x0095, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=14"

TRACE MINOR=0x0096, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=15"

TRACE MINOR=0x0097, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=16"

TRACE MINOR=0x0098, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=17"

TRACE MINOR=0x0099, TP=@STATIC, TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=18"

```

TRACE  MINOR=0x009A,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=19"

TRACE  MINOR=0x009B,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1A"

TRACE  MINOR=0x009C,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1B"

TRACE  MINOR=0x009D,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1C"

TRACE  MINOR=0x009E,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1D"

TRACE  MINOR=0x009F,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1E"

TRACE  MINOR=0x00A0,TP=@STATIC,TYPE=(POST),
DESC="(OS) Hardware Interrupt Post-Invocation",
FMT ="Interrupt Level=1F"

```

FAT file system operations

***DT preLockOper - w_LockOper pre-invocation internal trace point

```

TRACE MINOR = <RAS_PRE_INT_LOCKOPER>,
TP      = .w_LockOper,
TYPE    = (PRE,INT),
GROUP   = FS,
DESC    = "(OS) File Lock/Unlock Pre-Invocation",
REGS    = (AX,BX,DX,CX,DI,SI),
FMT     = "Unlock Flag = <FMT_WORD>  Handle = <FMT_WORD>",
FMT     = "Region Offset = <FMT_DWORD>  Length = <FMT_DWORD>"

```

***DT Allocate - Cluster Allocate internal trace points.

```

TRACE MINOR = <RAS_PRE_INT_ALLOCATE>,
TP      = .Allocate,
TYPE    = (PRE,INT),
GROUP   = FS,
DESC    = "(OS) Cluster Allocate Pre-Invocation",
REGS    = (CX),
FMT     = "Cluster Count Wanted = <FMT_WORD>"

TRACE MINOR = <RAS_POST_INT_ALLOCATE>,
TP      = .postAllocate,
TYPE    = (POST,INT),
GROUP   = FS,
DESC    = "(OS) Cluster Allocate Post-Invocation",
REGS    = (CX),
FMT     = "Cluster Count Obtained = <FMT_WORD>"

```

***DT preClusterRelease - Cluster Release pre-invocation internal trace point

```

TRACE MINOR = <RAS_PRE_INT_RELEASE>,
TP      = .preRelease,

```

```

TYPE = (PRE, INT),
GROUP = FS,
DESC = "(OS) Cluster Release Pre-Invocation",
REGS = (BX),
FMT = "Cluster Release = <FMT_WORD>"

```

File system operations (1/2)

***DT preDOS32ISETFHSTATE - Dos32ISetFHState pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32ISETFHSTATE>,
TP = .preDOS32ISETFHSTATE,
TYPE = (PRE, API),
GROUP = FS,
DESC = "(OS) Dos32ISetFHState Pre-Invocation",
<LOG_BX>,
<LOG_CX>,
<LOG_DX>,
FMT = "File Handle = <FMT_BX> State = <FMT_DX> <FMT_CX>"

```

***DT postDOS32ISETFHSTATE - Dos32ISetFHState post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32ISETFHSTATE>,
TP = .postDOS32ISETFHSTATE,
TYPE = (POST, API),
GROUP = FS,
DESC = "(OS) Dos32ISetFHState Post-Invocation",
<LOG_RETVAL>,
FMT = " Return Code = <FMT_RETVAL>"

```

***DT preDOS32IQUERYFHSTATE - Dos32IQUERYFHSTATE pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32IQUERYFHSTATE>,
TP = .preDOS32IQUERYFHSTATE,
TYPE = (PRE, API),
GROUP = FS,
DESC = "(OS) Dos32IQUERYFHSTATE Pre-Invocation",
<LOG_BX>,
FMT = "File Handle = <FMT_BX>"

```

***DT postDOS32IQUERYFHSTATE - Dos32IQUERYFHSTATE post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32IQUERYFHSTATE>,
TP = .postDOS32IQUERYFHSTATE,
TYPE = (POST, API),
GROUP = FS,
DESC = "(OS) Dos32IQUERYFHSTATE Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT = "Handle State = <FMT_CX> Return Code = <FMT_RETVAL>"

```

***DT preDOS32IPROTECTSETFHSTATE - Dos32IProtectSetFHState pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32IPROTECTSETFHSTATE>,
TP = .preDOS32IPROTECTSETFHSTATE,
TYPE = (PRE, API),
GROUP = FS,

```

```

DESC = "(OS) Dos32IProtectSetFHState Pre-Invocation",
<LOG_BX>,
<LOG_CX>,
<LOG_DX>,
FMT   = "File Handle = <FMT_BX>  State = <FMT_DX> <FMT_CX>"

```

*****DT postDOS32IPROTECTSETFHSTATE - Dos32IProtectSetFHState post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOS32IPROTECTSETFHSTATE>,
TP          = .postDOS32IPROTECTSETFHSTATE,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) Dos32IProtectSetFHState Post-Invocation",
<LOG_RETVAL>,
FMT         = "  Return Code = <FMT_RETVAL>"

```

*****DT preDOS32IPROTECTQUERYFHSTATE - Dos32IPROTECTQUERYFHSTATE pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOS32IPROTECTQUERYFHSTATE>,
TP          = .preDOS32IPROTECTQUERYFHSTATE,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) Dos32IPROTECTQUERYFHSTATE Pre-Invocation",
<LOG_BX>,
FMT         = "File Handle = <FMT_BX>"

```

*****DT postDOS32IPROTECTQUERYFHSTATE - Dos32IPROTECTQUERYFHSTATE post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOS32IPROTECTQUERYFHSTATE>,
TP          = .postDOS32IPROTECTQUERYFHSTATE,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) Dos32IPROTECTQUERYFHSTATE Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT         = "Handle State = <FMT_CX>  Return Code = <FMT_RETVAL>"

```

*****DT preDOSBUFRESET - DosBufReset pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSBUFRESET>,
TP          = .preDOSBUFRESET,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosBufReset Pre-Invocation",
<LOG_BX>,
FMT         = "Handle = <FMT_BX>"

```

*****DT postDOSBUFRESET - DosBufReset post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSBUFRESET>,
TP          = .postDOSBUFRESET,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosBufReset Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"

```

*****DT preDOSCHDIR - DosChDir pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSCHDIR>,
TP          = .preDOSCHDIR,
TYPE        = (PRE, API),

```

```

GROUP = FS,
DESC = "(OS) DosChDir Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT = "Path = <FMT_ASCIIIZ>"

```

***DT postDOSCHDIR - DosChDir post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCHDIR>,
TP = .postDOSCHDIR,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosChDir Post-Invocation",
<LOG_RETVAL>,
FMT = "Return code = <FMT_RETVAL>"

```

***DT preDOSCHGFILEPTR - DosChgFilePtr pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCHGFILEPTR>,
TP = .preDOSCHGFILEPTR,
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) DosChgFilePtr Pre-Invocation",
<LOG_AX>,
<LOG_CX>,
<LOG_DX>,
<LOG_BX>,
FMT = "Type = <FMT_AX> Distance = <FMT_CX><FMT_DX>"
FMT = "Handle = <FMT_BX>"

```

***DT postDOSCHGFILEPTR - DosChgFilePtr post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCHGFILEPTR>,
TP = .postDOSCHGFILEPTR,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosChgFilePtr Post-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT = "Location = <FMT_DX><FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSPROTECTCHGFILEPTR - DosProtectChgFilePtr pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSPROTECTCHGFILEPTR>,
TP = .preDOSPROTECTCHGFILEPTR,
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) DosProtectChgFilePtr Pre-Invocation",
<LOG_AX>,
<LOG_CX>,
<LOG_DX>,
<LOG_BX>,
FMT = "Type = <FMT_AX> Distance = <FMT_CX><FMT_DX>"
FMT = "Handle = <FMT_BX>"

```

***DT postDOSPROTECTCHGFILEPTR - DosProtectChgFilePtr post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSPROTECTCHGFILEPTR>,
TP = .postDOSPROTECTCHGFILEPTR,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosProtectChgFilePtr Post-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_RETVAL>,

```

```
FMT    = "Location = <FMT_DX><FMT_AX> Return code = <FMT_RETVAL>"
```

*****DT preDOSCLOSE - DosClose pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSCLOSE>,  
TP          = .preDOSCLOSE,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosClose Pre-Invocation",  
<LOG_BX>,  
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSCLOSE - DosClose post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSCLOSE>,  
TP          = .postDOSCLOSE,  
TYPE        = (POST,API),  
GROUP       = FS,  
DESC        = "(OS) DosClose Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTCLOSE - DosProtectClose pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTCLOSE>,  
TP          = .preDOSPROTECTCLOSE,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosProtectClose Pre-Invocation",  
<LOG_BX>,  
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSPROTECTCLOSE - DosProtectClose post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTCLOSE>,  
TP          = .postDOSPROTECTCLOSE,  
TYPE        = (POST,API),  
GROUP       = FS,  
DESC        = "(OS) DosProtectClose Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSCLOSECHANGENOTIFY - DosCloseChangeNotify invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSCLOSECHANGENOTIFY>,  
TP          = .preDOSCLOSECHANGENOTIFY,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosCloseChangeNotify Pre-Invocation",  
<LOG_BX>,  
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSCLOSECHANGENOTIFY - DosCloseChangeNotify post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSCLOSECHANGENOTIFY>,  
TP          = .postDOSCLOSECHANGENOTIFY,  
TYPE        = (POST,API),  
GROUP       = FS,  
DESC        = "(OS) DosCloseChangeNotify Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return code = <FMT_RETVAL>"
```

***DT preDOSDELETE - DosDelete pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSDELETE>,
TP          = .preDOSDELETE,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosDelete Pre-Invocation",
ASCIIIZ     = (<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT         = "Path = <FMT_ASCIIIZ>"
```

***DT postDOSDELETE - DosDelete post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSDELETE>,
TP          = .postDOSDELETE,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosDelete Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

***DT preDOSDEVICTL - DosDevIoctl pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSDEVICTL>,
TP          = .preDOSDEVICTL,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosDevIoctl Pre-Invocation",
<LOG_BX>,
MEM         = (<KSTK_ARGS>+<SKIP_WORD>,DIRECT,2),
MEM         = (<KSTK_ARGS>+<SKIP_WORD>+<SKIP_WORD>,DIRECT,2),
FMT         = "Handle = <FMT_BX> Category = %P %W Function = %P %W"
```

***DT postDOSDEVICTL - DosDevIoctl post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSDEVICTL>,
TP          = .postDOSDEVICTL,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosDevIoctl Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

***DT preDOSDEVICTL2 - DosDevIoctl2 pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSDEVICTL2>,
TP          = .preDOSDEVICTL2,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosDevIoctl2 Pre-Invocation",
<LOG_BX>,
MEM         = (<KSTK_ARGS>+<SKIP_WORD>,DIRECT,2),
MEM         = (<KSTK_ARGS>+<SKIP_WORD>+<SKIP_WORD>,DIRECT,2),
FMT         = "Handle = <FMT_BX> Category = %P %W Function = %P %W"
```

***DT postDOSDEVICTL2 - DosDevIoctl2 post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSDEVICTL2>,
TP          = .postDOSDEVICTL2,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosDevIoctl2 Post-Invocation",
<LOG_RETVAL>,
```

```
FMT    = "Return code = <FMT_RETVAL>"
```

*****DT preDOSDUPHANDLE - DosDupHandle pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSDUPHANDLE>,  
TP          = .preDOSDUPHANDLE,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosDupHandle Pre-Invocation",  
<LOG_BX>,  
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSDUPHANDLE - DosDupHandle post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSDUPHANDLE>,  
TP          = .postDOSDUPHANDLE,  
TYPE        = (POST,API),  
GROUP       = FS,  
DESC        = "(OS) DosDupHandle Post-Invocation",  
<LOG_AX>,  
<LOG_RETVAL>,  
FMT         = "New handle = <FMT_AX> Return code = <FMT_RETVAL>"
```

*****DT preDOSEEDITNAME - DosEditName pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSEEDITNAME>,  
TP          = .preDOSEEDITNAME,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosEditName Pre-Invocation",  
ASCIIIZ= (<KSTK_ARGS>+6,I,<SHORTPATHLEN>),  
ASCIIIZ= (<KSTK_ARGS>+10,I,<SHORTPATHLEN>),  
<LOG_BX>,  
FMT         = "Edit string = <FMT_ASCIIIZ>",  
FMT         = "Source string = <FMT_ASCIIIZ>",  
FMT         = "Level = <FMT_BX>"
```

*****DT postDOSEEDITNAME - DosEditName post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSEEDITNAME>,  
TP          = .postDOSEEDITNAME,  
TYPE        = (POST,API),  
GROUP       = FS,  
DESC        = "(OS) DosEditName Post-Invocation",  
ASCIIIZ= (<KSTK_ARGS>+<SKIP_WORD>,INDIRECT,<MAXPATHLEN>),  
<LOG_RETVAL>,  
FMT         = "Resultant string = <FMT_ASCIIIZ>",  
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSENUMATTRIBUTE - DosEnumAttribute pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSENUMATTRIBUTE>,  
TP          = .preDOSENUMATTRIBUTE,  
TYPE        = (PRE,API),  
GROUP       = FS,  
DESC        = "(OS) DosEnumAttribute Pre-Invocation",
```

*****DT postDOSENUMATTRIBUTE - DosEnumAttribute post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSENUMATTRIBUTE>,  
TP          = .postDOSENUMATTRIBUTE,  
TYPE        = (POST,API),
```

```
GROUP = FS,  
DESC = "(OS) DosEnumAttribute Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTENUMATTRIBUTE - DosProtectEnumAttribute pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTENUMATTRIBUTE>,  
TP = .preDOSPROTECTENUMATTRIBUTE,  
TYPE = (PRE,API),  
GROUP = FS,  
DESC = "(OS) DosProtectEnumAttribute Pre-Invocation",
```

*****DT postDOSPROTECTENUMATTRIBUTE - DosProtectEnumAttribute post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTENUMATTRIBUTE>,  
TP = .postDOSPROTECTENUMATTRIBUTE,  
TYPE = (POST,API),  
GROUP = FS,  
DESC = "(OS) DosProtectEnumAttribute Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFILEIO - DosFileIO pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFILEIO>,  
TP = .preDOSFILEIO,  
TYPE = (PRE,API),  
GROUP = FS,  
DESC = "(OS) DosFileIO Pre-Invocation",  
<LOG_BX>,  
FMT = "Handle = <FMT_BX>"
```

*****DT postDOSFILEIO - DosFileIO post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFILEIO>,  
TP = .postDOSFILEIO,  
TYPE = (POST,API),  
GROUP = FS,  
DESC = "(OS) DosFileIO Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTFILEIO - DosProtectFileIO pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTFILEIO>,  
TP = .preDOSPROTECTFILEIO,  
TYPE = (PRE,API),  
GROUP = FS,  
DESC = "(OS) DosProtectFileIO Pre-Invocation",  
<LOG_BX>,  
FMT = "Handle = <FMT_BX>"
```

*****DT postDOSPROTECTFILEIO - DosProtectFileIO post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTFILEIO>,  
TP = .postDOSPROTECTFILEIO,  
TYPE = (POST,API),  
GROUP = FS,  
DESC = "(OS) DosProtectFileIO Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFILELOCKS - DosFileLocks pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFILELOCKS>,
TP          = .preDOSFILELOCKS,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosFileLocks Pre-Invocation",
<LOG_BX>,
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSFILELOCKS - DosFileLocks post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFILELOCKS>,
TP          = .postDOSFILELOCKS,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosFileLocks Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTFILELOCKS - DosProtectFileLocks pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTFILELOCKS>,
TP          = .preDOSPROTECTFILELOCKS,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosProtectFileLocks Pre-Invocation",
<LOG_BX>,
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSPROTECTFILELOCKS - DosProtectFileLocks post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTFILELOCKS>,
TP          = .postDOSPROTECTFILELOCKS,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosProtectFileLocks Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFINDCLOSE - DosFindClose pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFINDCLOSE>,
TP          = .preDOSFINDCLOSE,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosFindClose Pre-Invocation",
<LOG_BX>,
FMT         = "Handle = <FMT_BX>"
```

*****DT postDOSFINDCLOSE - DosFindClose post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFINDCLOSE>,
TP          = .postDOSFINDCLOSE,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosFindClose Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFINDFIRST - DosFindFirst pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFINDFIRST>,
TP          = .preDOSFINDFIRST,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindFirst Pre-Invocation",
ASCIIIZ    = (<KSTK_ARGS>+20,INDIRECT,<MAXPATHLEN>),
FMT        = "Path = <FMT_ASCIIIZ>"
```

*****DT postDOSFINDFIRST - DosFindFirst post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFINDFIRST>,
TP          = .postDOSFINDFIRST,
TYPE       = (POST,API),
GROUP      = FS,
DESC       = "(OS) DosFindFirst Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT        = "Search handle = <FMT_BX> Return code = <FMT_RETVAL>"
```

*****DT preDOSFINDFIRST2 - DosFindFirst2 pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFINDFIRST2>,
TP          = .preDOSFINDFIRST2,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindFirst2 Pre-Invocation",
ASCIIIZ    = (<KSTK_ARGS>+22,INDIRECT,<MAXPATHLEN>),
FMT        = "Path = <FMT_ASCIIIZ>"
```

*****DT postDOSFINDFIRST2 - DosFindFirst2 post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFINDFIRST2>,
TP          = .postDOSFINDFIRST2,
TYPE       = (POST,API),
GROUP      = FS,
DESC       = "(OS) DosFindFirst2 Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT        = "Search handle = <FMT_BX> Return code = <FMT_RETVAL>"
```

*****DT preDOSFINDFROMNAME - DosFindFromName pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFINDFROMNAME>,
TP          = .preDOSFINDFROMNAME,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindFromName Pre-Invocation",
ASCIIIZ    = (<KSTK_ARGS>,INDIRECT,<MAXPATHLEN>),
MEM        = (<KSTK_ARGS>+<SKIP_PTR>,DIRECT,4),
MEM        = (<KSTK_ARGS>+18,DIRECT,2),
FMT        = "Name = <FMT_ASCIIIZ>",
FMT        = "Position = <FMT_C_PTR> Directory Handle = <FMT_C_WORD>"
```

*****DT postDOSFINDFROMNAME - DosFindFromName post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFINDFROMNAME>,
TP          = .postDOSFINDFROMNAME,
TYPE       = (POST,API),
GROUP      = FS,
DESC       = "(OS) DosFindFromName Post-Invocation",
<LOG_CX>,
```

```
<LOG_RETVAL>,
FMT    = "Search count = <FMT_CX> Return code = <FMT_RETVAL>"
```

***DT preDOSFINDNEXT - DosFindNext pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSFINDNEXT>,
TP          = .preDOSFINDNEXT,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindNext Pre-Invocation",
<LOG_BX>,
FMT        = "Handle = <FMT_BX>"
```

***DT postDOSFINDNEXT - DosFindNext post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSFINDNEXT>,
TP          = .postDOSFINDNEXT,
TYPE       = (POST,API),
GROUP      = FS,
DESC       = "(OS) DosFindNext Post-Invocation",
<LOG_RETVAL>,
FMT        = "Return code = <FMT_RETVAL>"
```

***DT preDOSFINDNOTIFYCLOSE - DosFindNotifyClose pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSFINDNOTIFYCLOSE>,
TP          = .preDOSFINDNOTIFYCLOSE,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindNotifyClose Pre-Invocation",
<LOG_BX>,
FMT        = "Handle = <FMT_BX>"
```

***DT postDOSFINDNOTIFYCLOSE - DosFindNotifyClose post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSFINDNOTIFYCLOSE>,
TP          = .postDOSFINDNOTIFYCLOSE,
TYPE       = (POST,API),
GROUP      = FS,
DESC       = "(OS) DosFindNotifyClose Post-Invocation",
<LOG_RETVAL>,
FMT        = "Return code = <FMT_RETVAL>"
```

***DT preDOSFINDNOTIFYFIRST - DosFindNotifyFirst pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSFINDNOTIFYFIRST>,
TP          = .preDOSFINDNOTIFYFIRST,
TYPE       = (PRE,API),
GROUP      = FS,
DESC       = "(OS) DosFindNotifyFirst Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+26,INDIRECT,<MAXPATHLEN>),
MEM        = (<KSTK_ARGS>+20,DIRECT,2),
MEM        = (<KSTK_ARGS>+8,DIRECT,2),
<LOG_DX>,
<LOG_AX>,
FMT        = "Name = <FMT_ASCIIIZ>",
FMT        = "Attribute = %P %W Level = %P %W",
FMT        = "Timeout = <FMT_DX><FMT_AX>"
```

***DT postDOSFINDNOTIFYFIRST - DosFindNotifyFirst post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSFINDNOTIFYFIRST>,
```

```

TP      = .postDOSFINDNOTIFYFIRST,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosFindNotifyFirst Post-Invocation",
<LOG_CX>,
<LOG_BX>,
<LOG_RETVAL>,
FMT      = "Search count = <FMT_CX> Handle = <FMT_BX>",
FMT      = "Return code = <FMT_RETVAL>"

```

*****DT preDOSFINDNOTIFYNEXT - DosFindNotifyNext pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSFINDNOTIFYNEXT>,
TP          = .preDOSFINDNOTIFYNEXT,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosFindNotifyNext Pre-Invocation",
<LOG_SI>,
<LOG_BX>,
FMT         = "Change count = <FMT_SI> Handle = <FMT_BX>"

```

*****DT postDOSFINDNOTIFYNEXT - DosFindNotifyNext post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSFINDNOTIFYNEXT>,
TP          = .postDOSFINDNOTIFYNEXT,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosFindNotifyNext Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT         = "Change count = <FMT_CX> Return code = <FMT_RETVAL>"

```

*****DT preDOSFORCEDELETE - DosForceDelete pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSFORCEDELETE>,
TP          = .preDOSFORCEDELETE,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosForceDelete Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT         = "Path = <FMT_ASCIIIZ>"

```

*****DT postDOSFORCEDELETE - DosForceDelete post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSFORCEDELETE>,
TP          = .postDOSFORCEDELETE,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosForceDelete Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"

```

*****DT preDOSFSATTACH - DosFSAttach pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSFSATTACH>,
TP          = .preDOSFSATTACH,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosFSAttach Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+16,INDIRECT,<SHORTPATHLEN>),
ASCIIIZ=(<KSTK_ARGS>+12,INDIRECT,<SHORTPATHLEN>),
<LOG_AX>,
FMT         = "Device = <FMT_ASCIIIZ>",
FMT         = "FSD = <FMT_ASCIIIZ>",
FMT         = "OpFlag = <FMT_AX>"

```

*****DT postDOSFSATTACH - DosFSAttach post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFSATTACH>,
TP      = .postDOSFSATTACH,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosFSAttach Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFSCTL - DosFSctl pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFSCTL>,
TP      = .preDOSFSCTL,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosFSctl Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+8,INDIRECT,<MAXPATHLEN>),
<LOG_CX>,
<LOG_AX>,
FMT     = "FSD = <FMT_ASCIIIZ>",
FMT     = "Handle = <FMT_CX> Route = <FMT_AX>"
```

*****DT postDOSFSCTL - DosFSctl post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFSCTL>,
TP      = .postDOSFSCTL,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosFSctl Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSICOPY - DosICopy pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSICOPY>,
TP      = .preDOSICOPY,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosICopy Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+10,INDIRECT,<SHORTPATHLEN>),
ASCIIIZ= (<KSTK_ARGS>+6,INDIRECT,<SHORTPATHLEN>),
<LOG_CX>,
FMT     = "Source = <FMT_ASCIIIZ>",
FMT     = "Target = <FMT_ASCIIIZ>",
FMT     = "OpMode = <FMT_CX>"
```

*****DT postDOSICOPY - DosICopy post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSICOPY>,
TP      = .postDOSICOPY,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosICopy Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSIDEVIOCTL - DosIDevioctl pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSIDEVIOCTL>,
TP      = .preDOSIDEVIOCTL,
```

```
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) DosDevIoctl Pre-Invocation",
```

*****DT postDOSIDEVIOCTL - DosDevIoctl post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSIDEVIOCTL>,
TP = .postDOSIDEVIOCTL,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosDevIoctl Post-Invocation",
<LOG_RETVAL>,
FMT = "Return code = <FMT_RETVAL>"
```

*****DT preDOSIREAD - DosRead pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSIREAD>,
TP = .preDOSIREAD,
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) DosRead Pre-Invocation",
<LOG_BX>,
<LOG_DS>,
<LOG_DX>,
<LOG_CX>,
<LOG_DI>,
<LOG_SI>,
FMT = "File Handle = <FMT_BX> Buffer = <FMT_DS>:<FMT_DX>",
FMT = "Buffer Size = <FMT_CX> Pointer to bytes read = <FMT_DI>:<FMT_SI>"
```

*****DT postDOSIREAD - DosRead post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSIREAD>,
TP = .postDOSIREAD,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosRead Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT = "Bytes Read = <FMT_CX> Return code = <FMT_RETVAL>"
```

*****DT preDOSIPROTECTREAD - DosProtectRead pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSIPROTECTREAD>,
TP = .preDOSIPROTECTREAD,
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) DosProtectRead Pre-Invocation",
<LOG_BX>,
FMT = "Handle = <FMT_BX>"
```

*****DT postDOSIPROTECTREAD - DosProtectRead post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSIPROTECTREAD>,
TP = .postDOSIPROTECTREAD,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) DosProtectRead Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT = "Bytes Read = <FMT_CX> Return code = <FMT_RETVAL>"
```

*****DT preDOSISETFILEINFO - DosSetFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSISETFILEINFO>,
TP      = .preDOSISETFILEINFO,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosISetFileInfo Pre-Invocation",
<LOG_DX>,
FMT     = "Info Level = <FMT_DX>"
```

*****DT postDOSISETFILEINFO - DosISetFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSISETFILEINFO>,
TP      = .postDOSISETFILEINFO,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosISetFileInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSIPROTECTSETFILEINFO - DosIProtectSetFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSIPROTECTSETFILEINFO>,
TP      = .preDOSIPROTECTSETFILEINFO,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosIProtectSetFileInfo Pre-Invocation",
<LOG_DX>,
FMT     = "Info Level = <FMT_DX>"
```

*****DT postDOSIPROTECTSETFILEINFO - DosIProtectSetFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSIPROTECTSETFILEINFO>,
TP      = .postDOSIPROTECTSETFILEINFO,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosIProtectSetFileInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSISETPATHINFO - DosSetPathInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSISETPATHINFO>,
TP      = .preDOSISETPATHINFO,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosISetPathInfo Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+14, INDIRECT, <MAXPATHLEN>),
<LOG_BX>,
FMT     = "Path =<FMT_ASCIIIZ>",
FMT     = "Info Level = <FMT_BX>"
```

*****DT postDOSISETPATHINFO - DosISetPathInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSISETPATHINFO>,
TP      = .postDOSISETPATHINFO,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosISetPathInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSISETRELMAXFH - DosISetRelMaxFH pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSISETRELMAXFH>,
TP      = .preDOSISETRELMAXFH,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosISetRelMaxFH Pre-Invocation",

```

***DT postDOSISETRELMAXFH - DosISetRelMaxFH post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSISETRELMAXFH>,
TP      = .postDOSISETRELMAXFH,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosISetRelMaxFH Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"

```

File system operations (2/2)

***DT preDOSIWRITE - DosIWrite pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSIWRITE>,
TP      = .preDOSIWRITE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosIWrite Pre-Invocation",
<LOG_BX>,
<LOG_DS>,
<LOG_DX>,
<LOG_CX>,
<LOG_DI>,
<LOG_SI>,
FMT     = "File Handle = <FMT_BX> Buffer = <FMT_DS>:<FMT_DX>",
FMT     = "Buffer Size = <FMT_CX> Pointer to bytes written = <FMT_DI>:<FMT_SI>"

```

***DT postDOSIWRITE - DosIWrite post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSIWRITE>,
TP      = .postDOSIWRITE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosIWrite Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Bytes Written = <FMT_CX> Return code = <FMT_RETVAL>"

```

***DT preDOSIPROTECTWRITE - DosIProtectWrite pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSIPROTECTWRITE>,
TP      = .preDOSIPROTECTWRITE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosIPROTECTWrite Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"

```

***DT postDOSIPROTECTWRITE - DosIProtectWrite post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSIPROTECTWRITE>,
TP      = .postDOSIPROTECTWRITE,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosIPProtectWrite Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Bytes Written = <FMT_CX> Return code = <FMT_RETVAL>"
```

***DT preDOSMKDIR - DosMkDir pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSMKDIR>,
TP      = .preDOSMKDIR,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosMkDir Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_DWORD>,INDIRECT,<MAXPATHLEN>),
FMT     = "Path = <FMT_ASCIIIZ>"
```

***DT postDOSMKDIR - DosMkDir post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSMKDIR>,
TP      = .postDOSMKDIR,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosMkDir Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

***DT preDOSMKDIR2 - DosMkDir2 pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSMKDIR2>,
TP      = .preDOSMKDIR2,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosMkDir2 Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_DWORD>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT     = "Path = <FMT_ASCIIIZ>"
```

***DT postDOSMKDIR2 - DosMkDir2 post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSMKDIR2>,
TP      = .postDOSMKDIR2,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosMkDir2 Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

***DT preDOSMOVE - DosMove pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSMOVE>,
TP      = .preDOSMOVE,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosMove Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_DWORD>,I,<SHORTPATHLEN>),
ASCIIIZ= (<KSTK_ARGS>+<SKIP_DWORD>+<SKIP_PTR>,I,<SHORTPATHLEN>),
FMT     = "New name = <FMT_ASCIIIZ>",
FMT     = "Old name = <FMT_ASCIIIZ>"
```

***DT postDOSMOVE - DosMove post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSMOVE>,
TP      = .postDOSMOVE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosMove Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSNEWSIZE - DosNewSize pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSNEWSIZE>,
TP      = .preDOSNEWSIZE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosNewSize Pre-Invocation",
<LOG_CX>,
<LOG_DX>,
<LOG_BX>,
FMT     = "Filesize <FMT_CX><FMT_DX> Handle = <FMT_BX>"

```

***DT postDOSNEWSIZE - DosNewSize post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSNEWSIZE>,
TP      = .postDOSNEWSIZE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosNewSize Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"

```

***DT preDOSPROTECTNEWSIZE - DosProtectNewSize pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSPROTECTNEWSIZE>,
TP      = .preDOSPROTECTNEWSIZE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosProtectNewSize Pre-Invocation",
<LOG_CX>,
<LOG_DX>,
<LOG_BX>,
FMT     = "Filesize <FMT_CX><FMT_DX> Handle = <FMT_BX>"

```

***DT postDOSPROTECTNEWSIZE - DosProtectNewSize post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSPROTECTNEWSIZE>,
TP      = .postDOSPROTECTNEWSIZE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosProtectNewSize Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"

```

***DT preDOSOPEN - DosOpen pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPEN>,
TP      = .preDOSOPEN,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosOpen Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+22, INDIRECT, <MAXPATHLEN>),
MEM     = (<KSTK_ARGS>+4, DIRECT, 2),
MEM     = (<KSTK_ARGS>+6, DIRECT, 2),

```

```

MEM  = (<KSTK_ARGS>+8, DIRECT, 2),
MEM  = (<KSTK_ARGS>+10,DIRECT, 4),
FMT  = "Filename = <FMT_ASCIIIZ>",
FMT  = "Mode = <FMT_C_WORD>  Control = <FMT_C_WORD>",
FMT  = "Attrib = <FMT_C_WORD>  Size = <FMT_C_PTR>"

```

***DT postDOSOPEN - DosOpen post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPEN>,
TP          = .postDOSOPEN,
TYPE       = (POST, API),
GROUP      = FS,
DESC       = "(OS) DosOpen Post-Invocation",
<LOG_BX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT        = "Action = <FMT_BX>  Handle = <FMT_AX>"
FMT        = "Return Code = <FMT_RETVAL>"

```

***DT preDOSOPEN2 - DosOpen pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPEN2>,
TP          = .preDOSOPEN2,
TYPE       = (PRE, API),
GROUP      = FS,
DESC       = "(OS) DosOpen2 Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+28, INDIRECT, <MAXPATHLEN>),
MEM        = (<KSTK_ARGS>+8, DIRECT, 2),
MEM        = (<KSTK_ARGS>+12,DIRECT, 2),
MEM        = (<KSTK_ARGS>+14,DIRECT, 2),
MEM        = (<KSTK_ARGS>+16,DIRECT, 4),
FMT        = "Filename = <FMT_ASCIIIZ>",
FMT        = "Mode = <FMT_C_WORD>  Control = <FMT_C_WORD>",
FMT        = "Attrib = <FMT_C_WORD>  Size = <FMT_C_DWORD>"

```

***DT postDOSOPEN2 - DosOpen2 post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPEN2>,
TP          = .postDOSOPEN2,
TYPE       = (POST, API),
GROUP      = FS,
DESC       = "(OS) DosOpen2 Post-Invocation",
<LOG_BX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT        = "Action = <FMT_BX>  Handle = <FMT_AX>"
FMT        = "Return Code = <FMT_RETVAL>"

```

***DT preDOSOPEN2COMPT - DosOpen2Compt pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPEN2COMPT>,
TP          = .preDOSOPEN2COMPT,
TYPE       = (PRE, API),
GROUP      = FS,
DESC       = "(OS) DosOpen2Compt Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+28, INDIRECT, <MAXPATHLEN>),
MEM        = (<KSTK_ARGS>+8, DIRECT, 2),
MEM        = (<KSTK_ARGS>+12,DIRECT, 2),
MEM        = (<KSTK_ARGS>+14,DIRECT, 2),
MEM        = (<KSTK_ARGS>+16,DIRECT, 4),
FMT        = "Filename = <FMT_ASCIIIZ>",
FMT        = "Mode = <FMT_C_WORD>  Control = <FMT_C_WORD>",
FMT        = "Attrib = <FMT_C_WORD>  Size = <FMT_C_DWORD>"

```

***DT postDOSOPEN2COMPT - DosOpen2Compt post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPEN2COMPT>,
TP      = .postDOSOPEN2COMPT,
TYPE    = (POST, API),
GROUP   = FS,
DESC     = "(OS) DosOpen2Compt Post-Invocation",
<LOG_BX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Action = <FMT_BX> Handle = <FMT_AX>"
FMT     = "Return Code = <FMT_RETVAL>"

```

***DT preDOSPROTECTOPEN - DosOpen pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSPROTECTOPEN>,
TP      = .preDOSPROTECTOPEN,
TYPE    = (PRE, API),
GROUP   = FS,
DESC     = "(OS) DOSPROTECTOPEN Pre-Invocation",
ASCIIZ= (<KSTK_ARGS>+28, INDIRECT, <MAXPATHLEN>),
MEM      = (<KSTK_ARGS>+8, DIRECT, 2),
MEM      = (<KSTK_ARGS>+12, DIRECT, 2),
MEM      = (<KSTK_ARGS>+14, DIRECT, 2),
MEM      = (<KSTK_ARGS>+16, DIRECT, 4),
FMT     = "Filename = <FMT_ASCIIZ>",
FMT     = "Mode = <FMT_C_WORD> Control = <FMT_C_WORD>",
FMT     = "Attrib = <FMT_C_WORD> Size = <FMT_C_DWORD>"

```

***DT postDOSPROTECTOPEN - DosProtectOpen post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSPROTECTOPEN>,
TP      = .postDOSPROTECTOPEN,
TYPE    = (POST, API),
GROUP   = FS,
DESC     = "(OS) DosProtectOpen Post-Invocation",
<LOG_BX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Action = <FMT_BX> Handle = <FMT_AX>"
FMT     = "Return Code = <FMT_RETVAL>"

```

***DT preDOSOPENCHANGENOTIFY - DosOpenChangeNotify pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPENCHANGENOTIFY>,
TP      = .preDOSOPENCHANGENOTIFY,
TYPE    = (PRE, API),
GROUP   = FS,
DESC     = "(OS) DosOpenChangeNotify Pre-Invocation",

```

***DT postDOSOPENCHANGENOTIFY - DosOpenChangeNotify post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPENCHANGENOTIFY>,
TP      = .postDOSOPENCHANGENOTIFY,
TYPE    = (POST, API),
GROUP   = FS,
DESC     = "(OS) DosOpenChangeNotify Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSOPLOCKRELEASE - DosOpLockRelease pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPLOCKRELEASE>,
TP      = .preDOSOPLOCKRELEASE,
TYPE    = (PRE, API),
GROUP   = FS,

```

```

DESC = "(OS) DosOpLockRelease Pre-Invocation",
<LOG_DS>,
<LOG_SI>,
<LOG_CX>,
FMT   = "Kernel key = <FMT_DS>:<FMT_SI>   Flags = <FMT_CX>"

```

***DT postDOSOPLOCKRELEASE - DosOpLockRelease post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPLOCKRELEASE>,
TP          = .postDOSOPLOCKRELEASE,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosOpLockRelease Post-Invocation",

```

***DT preDOSOPLOCKWAIT - DosOpLockWait pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPLOCKWAIT>,
TP          = .preDOSOPLOCKWAIT,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosOpLockWait Pre-Invocation",

```

***DT postDOSOPLOCKWAIT - DosOpLockWait post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPLOCKWAIT>,
TP          = .postDOSOPLOCKWAIT,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosOpLockWait Post-Invocation",
MEM         = (<KSTK_ARGS>, INDIRECT, 4),
MEM         = (<KSTK_ARGS>+4, INDIRECT, 4),
FMT         = "Server Key = <FMT_C_DWORD>   Kernel Key = <FMT_C_DWORD>"

```

***DT preDOSQCURDIR - DosQCurDir pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSQCURDIR>,
TP          = .preDOSQCURDIR,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosQCurDir Pre-Invocation",
<LOG_DX>,
FMT         = "Drive Number = <FMT_DX>"

```

***DT postDOSQCURDIR - DosQCurDir post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSQCURDIR>,
TP          = .postDOSQCURDIR,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosQCurDir Post-Invocation",
ASCIIIZ= (<KSTK_ARGS>+4, INDIRECT, <MAXPATHLEN>),
<LOG_CX>,
<LOG_RETVAL>,
FMT         = "Current Dir = <FMT_ASCIIIZ>",
FMT         = "Len = <FMT_CX>   Return Code = <FMT_RETVAL>"

```

***DT preDOSQCURDISK - DosQCurDisk pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSQCURDISK>,
TP          = .preDOSQCURDISK,
TYPE        = (PRE, API),
GROUP       = FS,

```

```
DESC = "(OS) DosQCurDisk Pre-Invocation",
```

*****DT postDOSQCURDISK - DosQCurDisk post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQCURDISK>,
TP      = .postDOSQCURDISK,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosQCurDisk Post-Invocation",
<LOG_BX>,
MEM     = (<KSTK_ARGS>, INDIRECT, 4),
<LOG_RETVAL>,
FMT     = "Default Drive = <FMT_BX> Logical Map = <FMT_C_DWORD>",
FMT     = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSQFHANDSTATE - DosQFHandState pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQFHANDSTATE>,
TP      = .preDOSQFHANDSTATE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosQFHandState Pre-Invocation",
<LOG_BX>,
FMT     = "File Handle = <FMT_BX>"
```

*****DT postDOSQFHANDSTATE - DosQFHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQFHANDSTATE>,
TP      = .postDOSQFHANDSTATE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosQFHandState Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Handle State = <FMT_CX> Return Code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTQFHANDSTATE - DosProtectQFHandState pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTQFHANDSTATE>,
TP      = .preDOSPROTECTQFHANDSTATE,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosProtectQFHandState Pre-Invocation",
<LOG_BX>,
FMT     = "File Handle = <FMT_BX>"
```

*****DT postDOSPROTECTQFHANDSTATE - DosProtectQFHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTQFHANDSTATE>,
TP      = .postDOSPROTECTQFHANDSTATE,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosProtectQFHandState Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Handle State = <FMT_CX> Return Code = <FMT_RETVAL>"
```

*****DT preDOSQFILEINFO - DosQFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQFILEINFO>,
TP      = .preDOSQFILEINFO,
TYPE    = (PRE, API),
```

```
GROUP = FS,  
DESC = "(OS) DosQFileInfo Pre-Invocation",  
<LOG_BX>,  
<LOG_DX>,  
FMT = "File Handle = <FMT_BX> Info Level = <FMT_DX>"
```

*****DT postDOSQFILEINFO - DosQFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQFILEINFO>,  
TP = .postDOSQFILEINFO,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosQFileInfo Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTQFILEINFO - DosProtectQFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTQFILEINFO>,  
TP = .preDOSPROTECTQFILEINFO,  
TYPE = (PRE, API),  
GROUP = FS,  
DESC = "(OS) DosProtectQFileInfo Pre-Invocation",  
<LOG_BX>,  
<LOG_DX>,  
FMT = "File Handle = <FMT_BX> Info Level = <FMT_DX>"
```

*****DT postDOSPROTECTQFILEINFO - DosProtectQFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTQFILEINFO>,  
TP = .postDOSPROTECTQFILEINFO,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosProtectQFileInfo Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSQFILEMODE - DosQFileMode pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQFILEMODE>,  
TP = .preDOSQFILEMODE,  
TYPE = (PRE, API),  
GROUP = FS,  
DESC = "(OS) DosQFileMode Pre-Invocation",  
ASCIIIZ= (<KSTK_ARGS>+8, INDIRECT, <MAXPATHLEN>),  
FMT = "File = <FMT_ASCIIIZ>"
```

*****DT postDOSQFILEMODE - DosQFileMode post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQFILEMODE>,  
TP = .postDOSQFILEMODE,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosQFileMode Post-Invocation",  
<LOG_CX>,  
<LOG_RETVAL>,  
FMT = "File Mode = <FMT_CX> Return Code = <FMT_RETVAL>"
```

*****DT preDOSQFSATTACH - DosQFSAttach pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQFSATTACH>,  
TP = .preDOSQFSATTACH,
```

```

TYPE = (PRE, API),
GROUP = FS,
DESC = "(OS) DosQFSAttach Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+16, INDIRECT, <MAXPATHLEN>),
<LOG_AX>,
FMT = "Path = <FMT_ASCIIIZ>",
FMT = "Info Level = <FMT_AX>"

```

***DT postDOSQFSATTACH - DosQFSAttach post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSQFSATTACH>,
TP = .postDOSQFSATTACH,
TYPE = (POST, API),
GROUP = FS,
DESC = "(OS) DosQFSAttach Post-Invocation",
<LOG_RETVAL>,
FMT = "Return Code = <FMT_RETVAL>"

```

***DT preDOSQFSINFO - DosQFSInfo pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSQFSINFO>,
TP = .preDOSQFSINFO,
TYPE = (PRE, API),
GROUP = FS,
DESC = "(OS) DosQFSInfo Pre-Invocation",
<LOG_DX>,
<LOG_AX>,
FMT = "Drive = <FMT_DX> Info Level = <FMT_AX>"

```

***DT postDOSQFSINFO - DosQFSInfo post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSQFSINFO>,
TP = .postDOSQFSINFO,
TYPE = (POST, API),
GROUP = FS,
DESC = "(OS) DosQFSInfo Post-Invocation",
<LOG_RETVAL>,
FMT = "Return Code = <FMT_RETVAL>"

```

***DT preDOSQHANDTYPE - DosQHandType pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSQHANDTYPE>,
TP = .preDOSQHANDTYPE,
TYPE = (PRE, API),
GROUP = FS,
DESC = "(OS) DosQHandType Pre-Invocation",
<LOG_BX>,
FMT = "Handle = <FMT_BX>"

```

***DT postDOSQHANDTYPE - DosQHandType post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSQHANDTYPE>,
TP = .postDOSQHANDTYPE,
TYPE = (POST, API),
GROUP = FS,
DESC = "(OS) DosQHandType Post-Invocation",
MEM = (<KSTK_ARGS>+4, INDIRECT, 2),
MEM = (<KSTK_ARGS>, INDIRECT, 2),
<LOG_RETVAL>,
FMT = "Handle Type = <FMT_C_WORD> Flags = <FMT_C_WORD>",
FMT = "Return Code = <FMT_RETVAL>"

```

***DT preDOSQPATHINFO - DosQPathInfo pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSQPATHINFO>,
TP      = .preDOSQPATHINFO,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosQPathInfo Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+12, INDIRECT, <MAXPATHLEN>),
<LOG_BX>,
FMT     = "Path = <FMT_ASCIIIZ>",
FMT     = "Info Level = <FMT_BX>"

```

*****DT postDOSQPATHINFO - DosQPathInfo post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSQPATHINFO>,
TP      = .postDOSQPATHINFO,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosQPathInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"

```

*****DT preDOSQSYSINFO - DosQSysInfo pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSQSYSINFO>,
TP      = .preDOSQSYSINFO,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosQSysInfo Pre-Invocation",
<LOG_SI>,
FMT     = "Index = <FMT_SI>"

```

*****DT postDOSQSYSINFO - DosQSysInfo post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSQSYSINFO>,
TP      = .postDOSQSYSINFO,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosQSysInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return Code = <FMT_RETVAL>"

```

*****DT preDOSQVERIFY - DosQVerify pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSQVERIFY>,
TP      = .preDOSQVERIFY,
TYPE    = (PRE, API),
GROUP   = FS,
DESC    = "(OS) DosQVerify Pre-Invocation",

```

*****DT postDOSQVERIFY - DosQVerify post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSQVERIFY>,
TP      = .postDOSQVERIFY,
TYPE    = (POST, API),
GROUP   = FS,
DESC    = "(OS) DosQVerify Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Verify Flag = <FMT_AX> Return Code = <FMT_RETVAL>"

```

*****DT preDOSRESETCHANGENOTIFY - DosResetChangeNotify pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSRESETECHANGENOTIFY>,
TP          = .preDOSRESETECHANGENOTIFY,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosResetChangeNotify Pre-Invocation",

```

*****DT postDOSRESETECHANGENOTIFY - DosResetChangeNotify post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSRESETECHANGENOTIFY>,
TP          = .postDOSRESETECHANGENOTIFY,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosResetChangeNotify Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"

```

*****DT preDOSRMDIR - DosRmdir pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSRMDIR>,
TP          = .preDOSRMDIR,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosRmdir Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+4, INDIRECT, <MAXPATHLEN>),
FMT         = "Directory = <FMT_ASCIIIZ>"

```

*****DT postDOSRMDIR - DosRmdir post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSRMDIR>,
TP          = .postDOSRMDIR,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosRmdir Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return Code = <FMT_RETVAL>"

```

*****DT preDOSSELECTDISK - DosSelectDisk pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSSELECTDISK>,
TP          = .preDOSSELECTDISK,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosSelectDisk Pre-Invocation",
<LOG_DX>,
FMT         = "Drive Number = <FMT_DX>"

```

*****DT postDOSSELECTDISK - DosSelectDisk post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSSELECTDISK>,
TP          = .postDOSSELECTDISK,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosSelectDisk Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return Code = <FMT_RETVAL>"

```

*****DT preDOSSETFHANDSTATE - DosSetFHandState pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSSETFHANDSTATE>,
TP          = .preDOSSETFHANDSTATE,
TYPE        = (PRE, API),

```

```
GROUP = FS,  
DESC = "(OS) DosSetFHandState Pre-Invocation",  
<LOG_BX>,  
<LOG_CX>,  
FMT = "File Handle = <FMT_BX> State = <FMT_CX>"
```

*****DT postDOSSETFHANDSTATE - DosSetFHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETFHANDSTATE>,  
TP = .postDOSSETFHANDSTATE,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosSetFHandState Post-Invocation",  
<LOG_RETVAL>,  
FMT = " Return Code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTSETFHANDSTATE - DosSetProtectFHandState pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTSETFHANDSTATE>,  
TP = .preDOSPROTECTSETFHANDSTATE,  
TYPE = (PRE, API),  
GROUP = FS,  
DESC = "(OS) DosProtectSetFHandState Pre-Invocation",  
<LOG_BX>,  
<LOG_CX>,  
FMT = "File Handle = <FMT_BX> State = <FMT_CX>"
```

*****DT postDOSPROTECTSETFHANDSTATE - DosProtectSetFHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTSETFHANDSTATE>,  
TP = .postDOSPROTECTSETFHANDSTATE,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosProtectSetFHandState Post-Invocation",  
<LOG_RETVAL>,  
FMT = " Return Code = <FMT_RETVAL>"
```

*****DT preDOSSETFILEINFO - DosSetFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETFILEINFO>,  
TP = .preDOSSETFILEINFO,  
TYPE = (PRE, API),  
GROUP = FS,  
DESC = "(OS) DosSetFileInfo Pre-Invocation",  
<LOG_DX>,  
FMT = "Info Level = <FMT_DX>"
```

*****DT postDOSSETFILEINFO - DosSetFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETFILEINFO>,  
TP = .postDOSSETFILEINFO,  
TYPE = (POST, API),  
GROUP = FS,  
DESC = "(OS) DosSetFileInfo Post-Invocation",  
<LOG_RETVAL>,  
FMT = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSPROTECTSETFILEINFO - DosProtectSetFileInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPROTECTSETFILEINFO>,  
TP = .preDOSPROTECTSETFILEINFO,  
TYPE = (PRE, API),
```

```
GROUP = FS,  
DESC  = "(OS) DosProtectSetFileInfo Pre-Invocation",  
<LOG_DX>,  
FMT   = "Info Level = <FMT_DX>"
```

*****DT postDOSPROTECTSETFILEINFO - DosSetProtectFileInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPROTECTSETFILEINFO>,  
TP          = .postDOSPROTECTSETFILEINFO,  
TYPE        = (POST, API),  
GROUP       = FS,  
DESC        = "(OS) DosProtectSetFileInfo Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSSETFILEMODE - DosSetFileMode pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETFILEMODE>,  
TP          = .preDOSSETFILEMODE,  
TYPE        = (PRE, API),  
GROUP       = FS,  
DESC        = "(OS) DosSetFileMode Pre-Invocation",  
ASCIIIZ     = (<KSTK_ARGS>+6, INDIRECT, <MAXPATHLEN>),  
FMT         = "File = <FMT_ASCIIIZ>"
```

*****DT postDOSSETFILEMODE - DosSetFileMode post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETFILEMODE>,  
TP          = .postDOSSETFILEMODE,  
TYPE        = (POST, API),  
GROUP       = FS,  
DESC        = "(OS) DosSetFileMode Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSSETFSINFO - DosSetFSInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETFSINFO>,  
TP          = .preDOSSETFSINFO,  
TYPE        = (PRE, API),  
GROUP       = FS,  
DESC        = "(OS) DosSetFSInfo Pre-Invocation",  
<LOG_AX>,  
FMT         = "Level = <FMT_AX>"
```

*****DT postDOSSETFSINFO - DosSetFSInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETFSINFO>,  
TP          = .postDOSSETFSINFO,  
TYPE        = (POST, API),  
GROUP       = FS,  
DESC        = "(OS) DosSetFSInfo Post-Invocation",  
<LOG_RETVAL>,  
FMT         = "Return Code = <FMT_RETVAL>"
```

*****DT preDOSSETMAXFH - DosSetMaxFH pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETMAXFH>,  
TP          = .preDOSSETMAXFH,  
TYPE        = (PRE, API),  
GROUP       = FS,  
DESC        = "(OS) DosSetMaxFH Pre-Invocation",
```

```
<LOG_BX>,
FMT    = "Handles = <FMT_BX>"
```

***DT postDOSSETMAXFH - DosSetMaxFH post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSETMAXFH>,
TP          = .postDOSSETMAXFH,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosSetMaxFH Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return Code = <FMT_RETVAL>"
```

***DT preDOSSETPATHINFO - DosSetPathInfo pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSSETPATHINFO>,
TP          = .preDOSSETPATHINFO,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosSetPathInfo Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+14, INDIRECT, <MAXPATHLEN>),
<LOG_BX>,
FMT         = "Path =<FMT_ASCIIIZ>",
FMT         = "Info Level = <FMT_BX>"
```

***DT postDOSSETPATHINFO - DosSetPathInfo post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSETPATHINFO>,
TP          = .postDOSSETPATHINFO,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosSetPathInfo Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return Code = <FMT_RETVAL>"
```

***DT preDOSSETVERIFY - DosSetVerify pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSSETVERIFY>,
TP          = .preDOSSETVERIFY,
TYPE        = (PRE, API),
GROUP       = FS,
DESC        = "(OS) DosSetVerify Pre-Invocation",
```

***DT postDOSSETVERIFY - DosSetVerify post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSETVERIFY>,
TP          = .postDOSSETVERIFY,
TYPE        = (POST, API),
GROUP       = FS,
DESC        = "(OS) DosSetVerify Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return Code = <FMT_RETVAL>"
```

Memory operations

*****DT preDOSALLOC HUGE - DosAllocHuge pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSALLOC HUGE>,
TP          = .preDOSALLOC HUGE,
TYPE        = (PRE,API),
GROUP       = SEL,
DESC        = "(OS) DosAllocHuge Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
<LOG_CX>,
FMT         = "Initial size=<FMT_AX><FMT_BX>  Max size = <FMT_CX>0000"
```

*****DT postDOSALLOC HUGE - DosAllocHuge post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSALLOC HUGE>,
TP          = .postDOSALLOC HUGE,
TYPE        = (POST,API),
GROUP       = SEL,
DESC        = "(OS) DosAllocHuge Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT         = "Base selector = <FMT_AX>  Return code = <FMT_RETVAL>"
```

*****DT preDOSALLOC SEG - DosAllocSeg pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSALLOC SEG>,
TP          = .preDOSALLOC SEG,
TYPE        = (PRE,API),
GROUP       = SEL,
DESC        = "(OS) DosAllocSeg Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT         = "Flags=<FMT_AX>  Size=<FMT_BX>"
```

*****DT postDOSALLOC SEG - DosAllocSeg post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSALLOC SEG>,
TP          = .postDOSALLOC SEG,
TYPE        = (POST,API),
GROUP       = SEL,
DESC        = "(OS) DosAllocSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT         = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"
```

*****DT preDOSALLOC SHR SEG - DosAllocShrSeg pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSALLOC SHR SEG>,
TP          = .preDOSALLOC SHR SEG,
TYPE        = (PRE,API),
GROUP       = SEL,
DESC        = "(OS) DosAllocShrSeg Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
<LOG_BX>,
FMT         = "Name = <FMT_ASCIIIZ>"
FMT         = "Size = <FMT_BX>"
```

*****DT postDOSALLOC SHR SEG - DosAllocShrSeg post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSALLOC SHR SEG>,
TP          = .postDOSALLOC SHR SEG,
TYPE        = (POST,API),
GROUP       = SEL,
```

```

DESC = "(OS) DosAllocShrSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT  = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"

```

***DT preDOSALLOCPROTHUGE - DosAllocProtHuge pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSALLOCPROTHUGE>,
TP      = .preDOSALLOCPROTHUGE,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocProtHuge Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
<LOG_CX>,
FMT     = "Initial size=<FMT_AX><FMT_BX>  Max size = <FMT_CX>0000"

```

***DT postDOSALLOCPROTHUGE - DosAllocProtHuge post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSALLOCPROTHUGE>,
TP      = .postDOSALLOCPROTHUGE,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocProtHuge Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Base selector = <FMT_AX>  Return code = <FMT_RETVAL>"

```

***DT preDOSALLOCPROTSEG - DosAllocProtSeg pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSALLOCPROTSEG>,
TP      = .preDOSALLOCPROTSEG,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocProtSeg Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT     = "Flags=<FMT_AX>  Size=<FMT_BX>"

```

***DT postDOSALLOCPROTSEG - DosAllocProtSeg post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSALLOCPROTSEG>,
TP      = .postDOSALLOCPROTSEG,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocProtSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"

```

***DT preDOSALLOCshrPROTSEG - DosAllocShrProtSeg pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSALLOCshrPROTSEG>,
TP      = .preDOSALLOCshrPROTSEG,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocShrProtSeg Pre-Invocation",
ASCIIZ=(<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
<LOG_BX>,
FMT     = "Name = <FMT_ASCIIZ>",
FMT     = "Size = <FMT_BX>"

```

***DT postDOSALLOCshrPROTSEG - DosAllocShrProtSeg post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSALLOCshrPROTSEG>,
TP      = .postDOSALLOCshrPROTSEG,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosAllocShrProtSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Selector = <FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSCLOSESEM - DosCloseSem pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCLOSESEM>,
TP      = .preDOSCLOSESEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosCloseSem Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT     = "Semaphore handle = <FMT_AX><FMT_BX>"

```

***DT postDOSCLOSESEM - DosCloseSem post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCLOSESEM>,
TP      = .postDOSCLOSESEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) DosCloseSem Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return value = <FMT_RETVAL>"

```

***DT preDOSCREATECSALIAS - DosCreateCSAlias pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCREATECSALIAS>,
TP      = .preDOSCREATECSALIAS,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosCreateCSAlias Pre-Invocation",
<LOG_AX>,
FMT     = "Data selector = <FMT_AX>"

```

***DT postDOSCREATECSALIAS - DosCreateCSAlias post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCREATECSALIAS>,
TP      = .postDOSCREATECSALIAS,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosCreateCSAlias Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Code selector = <FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSCREATESEM - DosCreateSem pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCREATESEM>,
TP      = .preDOSCREATESEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosCreateSem Pre-Invocation",
ASCIIIZ=<KSTK_ARGS>,INDIRECT,<MAXPATHLEN>),
FMT     = "Name = <FMT_ASCIIIZ>"

```

***DT postDOSCREATESEM - DosCreateSem post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCREATESEM>,
TP      = .postDOSCREATESEM,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosCreateSem Post-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Handle = <FMT_DX><FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSCSWAIT - DosCWait pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCSWAIT>,
TP      = .preDOSCSWAIT,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosCWait Pre-Invocation",
<LOG_CX>,
<LOG_BX>,
<LOG_AX>,
FMT     = "Pid = <FMT_CX> Wait = <FMT_BX> Action = <FMT_AX>"

```

***DT postDOSCSWAIT - DosCWait post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCSWAIT>,
TP      = .postDOSCSWAIT,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) DosCWait Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT     = "Pid = <FMT_BX> Return code = <FMT_RETVAL>"

```

***DT preDOSDEVCONFIG - DosDevConfig pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSDEVCONFIG>,
TP      = .preDOSDEVCONFIG,
TYPE    = (PRE,API),
GROUP   = FS,
DESC    = "(OS) DosDevConfig Pre-Invocation",
<LOG_CX>,
<LOG_BX>,
FMT     = "Parm = <FMT_CX> Item = <FMT_BX>"

```

***DT postDOSDEVCONFIG - DosDevConfig post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSDEVCONFIG>,
TP      = .postDOSDEVCONFIG,
TYPE    = (POST,API),
GROUP   = FS,
DESC    = "(OS) DosDevConfig Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSERROR - DosError pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSERROR>,
TP      = .preDOSERROR,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosError Pre-Invocation",
<LOG_AX>,

```

```
FMT    = "Error setting = <FMT_AX>"
```

*****DT postDOSERROR - DosError post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSERROR>,
TP          = .postDOSERROR,
TYPE        = (POST,API),
GROUP       = TK,
DESC        = "(OS) DosError Pre-Invocation",
<LOG_RETVAL>,
FMT         = "Return value = <FMT_RETVAL>"
```

*****DT preDOSFLAGPROCESS - DosFlagProcess pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFLAGPROCESS>,
TP          = .preDOSFLAGPROCESS,
TYPE        = (PRE,API),
GROUP       = SIG,
DESC        = "(OS) DosFlagProcess Pre-Invocation",
<LOG_DX>,
<LOG_BX>,
<LOG_AX>,
<LOG_CX>,
FMT         = "Pid = <FMT_DX> Action = <FMT_BX>",
FMT         = "Signal = <FMT_AX> Arg = <FMT_CX>"
```

*****DT postDOSFLAGPROCESS - DosFlagProcess post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFLAGPROCESS>,
TP          = .postDOSFLAGPROCESS,
TYPE        = (POST,API),
GROUP       = SIG,
DESC        = "(OS) DosFlagProcess Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFREERESOURCE - DosFreeResource pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFREERESOURCE>,
TP          = .preDOSFREERESOURCE,
TYPE        = (PRE,API),
GROUP       = LDR,
DESC        = "(OS) DosFreeResource Pre-Invocation",
<LOG_AX>,
FMT         = "Selector = <FMT_AX>"
```

*****DT postDOSFREERESOURCE - DosFreeResource post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSFREERESOURCE>,
TP          = .postDOSFREERESOURCE,
TYPE        = (POST,API),
GROUP       = LDR,
DESC        = "(OS) DosFreeResource Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSFREESEG - DosFreeSeg pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSFREESEG>,
TP          = .preDOSFREESEG,
TYPE        = (PRE,API),
GROUP       = SEL,
```

```
DESC = "(OS) DosFreeSeg Pre-Invocation",
<LOG_AX>,
FMT = "Selector = <FMT_AX>"
```

***DT postDOSFREESEG - DosFreeSeg post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSFREESEG>,
TP = .postDOSFREESEG,
TYPE = (POST,API),
GROUP = SEL,
DESC = "(OS) DosFreeSeg Post-Invocation",
<LOG_RETVAL>,
FMT = "Return code = <FMT_RETVAL>"
```

***DT preDOSGETMODHANDLE - DosGetModHandle pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETMODHANDLE>,
TP = .preDOSGETMODHANDLE,
TYPE = (PRE,API),
GROUP = LDR,
DESC = "(OS) DosGetModHandle Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT = "Module name = <FMT_ASCIIIZ>"
```

***DT postDOSGETMODHANDLE - DosGetModHandle post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETMODHANDLE>,
TP = .postDOSGETMODHANDLE,
TYPE = (POST,API),
GROUP = LDR,
DESC = "(OS) DosGetModHandle Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT = "Module handle = <FMT_BX> Return code = <FMT_RETVAL>"
```

***DT preDOSGETMODNAME - DosGetModName pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETMODNAME>,
TP = .preDOSGETMODNAME,
TYPE = (PRE,API),
GROUP = LDR,
DESC = "(OS) DosGetModName Pre-Invocation",
<LOG_AX>,
FMT = "Module handle = <FMT_AX>"
```

***DT postDOSGETMODNAME - DosGetModName post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETMODNAME>,
TP = .postDOSGETMODNAME,
TYPE = (POST,API),
GROUP = LDR,
DESC = "(OS) DosGetModName Post-Invocation",
ASCIIIZ= (<KSTK_ARGS>,INDIRECT,<MAXPATHLEN>),
<LOG_RETVAL>,
FMT = "Module name = <FMT_ASCIIIZ>",
FMT = "Return code = <FMT_RETVAL>"
```

***DT preDOSGETPID - DosGetPid pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETPID>,
TP = .preDOSGETPID,
TYPE = (PRE,API),
```

```
GROUP = TK,  
DESC  = "(OS) DosGetPid Pre-Invocation",
```

***DT postDOSGETPID - DosGetPid post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETPID>,  
TP          = .postDOSGETPID,  
TYPE        = (POST,API),  
GROUP       = TK,  
DESC        = "(OS) DosGetPid Post-Invocation",  
<LOG_AX>,  
<LOG_BX>,  
<LOG_CX>,  
FMT         = "PID = <FMT_AX>  TID = <FMT_BX>  PPID = <FMT_CX>"
```

***DT preDOSGETPROCADDR - DosGetProcAddr pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETPROCADDR>,  
TP          = .preDOSGETPROCADDR,  
TYPE        = (PRE,API),  
GROUP       = LDR,  
DESC        = "(OS) DosGetProcAddr Pre-Invocation",  
<LOG_AX>,  
FMT         = "Module handle = <FMT_AX>"
```

***DT postDOSGETPROCADDR - DosGetProcAddr post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETPROCADDR>,  
TP          = .postDOSGETPROCADDR,  
TYPE        = (POST,API),  
GROUP       = LDR,  
DESC        = "(OS) DosGetProcAddr Post-Invocation",  
<LOG_DX>,  
<LOG_AX>,  
<LOG_RETVAL>,  
FMT         = "Proc addr = <FMT_DX>:<FMT_AX>  Return code = <FMT_RETVAL>"
```

***DT preDOSGETRESOURCE - DosGetResource pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETRESOURCE>,  
TP          = .preDOSGETRESOURCE,  
TYPE        = (PRE,API),  
GROUP       = LDR,  
DESC        = "(OS) DosGetResource Pre-Invocation",  
<LOG_DX>,  
<LOG_AX>,  
<LOG_SI>,  
FMT         = "NameID = <FMT_DX>  Type = <FMT_AX>",  
FMT         = "Module handle = <FMT_SI>"
```

***DT postDOSGETRESOURCE - DosGetResource post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETRESOURCE>,  
TP          = .postDOSGETRESOURCE,  
TYPE        = (POST,API),  
GROUP       = LDR,  
DESC        = "(OS) DosGetResource Post-Invocation",  
<LOG_AX>,  
<LOG_RETVAL>,  
FMT         = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"
```

***DT preDOSGETRESOURCE2 - DosGetResource2 pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSGETRESOURCE2>,
TP      = .preDOSGETRESOURCE2,
TYPE    = (PRE,API),
GROUP   = LDR,
DESC    = "(OS) DosGetResource2 Pre-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_SI>,
FMT     = "NameID = <FMT_DX>  Type = <FMT_AX>",
FMT     = "Module handle = <FMT_SI>"

```

***DT postDOSGETRESOURCE2 - DosGetResource2 post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSGETRESOURCE2>,
TP      = .postDOSGETRESOURCE2,
TYPE    = (POST,API),
GROUP   = LDR,
DESC    = "(OS) DosGetResource2 Post-Invocation",
<LOG_AX>,
<LOG_DX>,
<LOG_RETVAL>,
FMT     = "Address = <FMT_AX>:<FMT_DX> Return code = <FMT_RETVAL>"

```

***DT preDOSGETSEG - DosGetSeg pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSGETSEG>,
TP      = .preDOSGETSEG,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosGetSeg Pre-Invocation",
<LOG_AX>,
FMT     = "Selector = <FMT_AX>"

```

***DT postDOSGETSEG - DosGetSeg post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSGETSEG>,
TP      = .postDOSGETSEG,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosGetSeg Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSGETSHRSEG - DosGetShrSeg pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSGETSHRSEG>,
TP      = .preDOSGETSHRSEG,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosGetShrSeg Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT     = "Name = <FMT_ASCIIIZ>"

```

***DT postDOSGETSHRSEG - DosGetShrSeg post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSGETSHRSEG>,
TP      = .postDOSGETSHRSEG,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosGetShrSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"

```

*****DT preDOSGETVERSION - DosGetVersion pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSGETVERSION>,
TP          = .preDOSGETVERSION,
TYPE       = (PRE,API),
GROUP      = TK,
DESC       = "(OS) DosGetVersion Pre-Invocation",
```

*****DT postDOSGETVERSION - DosGetVersion post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSGETVERSION>,
TP          = .postDOSGETVERSION,
TYPE       = (POST,API),
GROUP      = TK,
DESC       = "(OS) DosGetVersion Post-Invocation",
<LOG_AX>,
FMT        = "Major/minor version = <FMT_AX>"
```

*****DT preDOSGIVESEG - DosGiveSeg pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSGIVESEG>,
TP          = .preDOSGIVESEG,
TYPE       = (PRE,API),
GROUP      = SEL,
DESC       = "(OS) DosGiveSeg Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT        = "Selector = <FMT_AX>  Pid = <FMT_BX>"
```

*****DT postDOSGIVESEG - DosGiveSeg post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSGIVESEG>,
TP          = .postDOSGIVESEG,
TYPE       = (POST,API),
GROUP      = SEL,
DESC       = "(OS) DosGiveSeg Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT        = "Selector = <FMT_AX>  Return code = <FMT_RETVAL>"
```

*****DT preDOSIEXECPGM - DosIExecPgm pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSIEXECPGM>,
TP          = .preDOSIEXECPGM,
TYPE       = (PRE,API),
GROUP      = TK,
DESC       = "(OS) DosIExecPgm Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>,INDIRECT,<MAXPATHLEN>),
<LOG_AX>,
FMT        = "Program name = <FMT_ASCIIIZ>"
FMT        = "Exec flag = <FMT_AX>"
```

*****DT postDOSIEXECPGM - DosIExecPgm post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSIEXECPGM>,
TP          = .postDOSIEXECPGM,
TYPE       = (POST,API),
GROUP      = TK,
DESC       = "(OS) DosIExecPgm Post-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_RETVAL>,
```

```
FMT    = "Pid/Term Code = <FMT_DX>  Result Code = <FMT_AX>",
FMT    = "Return code = <FMT_RETVAL>"
```

*****DT preDOSKILLPROCESS - DosKillProcess pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSKILLPROCESS>,
TP          = .preDOSKILLPROCESS,
TYPE       = (PRE,API),
GROUP      = SIG,
DESC       = "(OS) DosKillProcess Pre-Invocation",
<LOG_BX>,
FMT        = "PID = <FMT_BX>"
```

*****DT postDOSKILLPROCESS - DosKillProcess post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSKILLPROCESS>,
TP          = .postDOSKILLPROCESS,
TYPE       = (POST,API),
GROUP      = SIG,
DESC       = "(OS) DosKillProcess Post-Invocation",
<LOG_RETVAL>,
FMT        = "Return code = <FMT_RETVAL>"
```

*****DT preDOSLOADMODULE - DosLoadModule pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSLOADMODULE>,
TP          = .preDOSLOADMODULE,
TYPE       = (PRE,API),
GROUP      = LDR,
DESC       = "(OS) DosLoadModule Pre-Invocation",
ASCIIIZ= (<KSTK_ARGS>+<SKIP_PTR>,INDIRECT,<MAXPATHLEN>),
FMT        = "Module name = <FMT_ASCIIIZ>"
```

*****DT postDOSLOADMODULE - DosLoadModule post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSLOADMODULE>,
TP          = .postDOSLOADMODULE,
TYPE       = (POST,API),
GROUP      = LDR,
DESC       = "(OS) DosLoadModule Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT        = "Module handle = <FMT_BX> Return code = <FMT_RETVAL>"
```

*****DT preDOSMAKEPIPE - DosMakePipe pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSMAKEPIPE>,
TP          = .preDOSMAKEPIPE,
TYPE       = (PRE,API),
GROUP      = PIP,
DESC       = "(OS) DosMakePipe Pre-Invocation",
<LOG_CX>,
FMT        = "Size of pipe = <FMT_CX>"
```

*****DT postDOSMAKEPIPE - DosMakePipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSMAKEPIPE>,
TP          = .postDOSMAKEPIPE,
TYPE       = (POST,API),
GROUP      = PIP,
DESC       = "(OS) DosMakePipe Post-Invocation",
<LOG_AX>,
```

```

<LOG_BX>,
<LOG_RETVAL>,
FMT    = "Read handle = <FMT_AX> Write handle = <FMT_BX>",
FMT    = "Return code = <FMT_RETVAL>"

```

***DT preDOSOPENSEM - DosOpenSem pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSOPENSEM>,
TP          = .preDOSOPENSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) DosOpenSem Pre-Invocation",
ASCIIZ      = (<KSTK_ARGS>,INDIRECT,<MAXPATHLEN>),
FMT         = "Name = <FMT_ASCIIZ>"

```

***DT postDOSOPENSEM - DosOpenSem post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSOPENSEM>,
TP          = .postDOSOPENSEM,
TYPE        = (POST,API),
GROUP       = SEM,
DESC        = "(OS) DosOpenSem Post-Invocation",
<LOG_DX>,
<LOG_AX>,
<LOG_RETVAL>,
FMT         = "Handle = <FMT_DX><FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSPHYSICALDISK - DosPhysicalDisk pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSPHYSICALDISK>,
TP          = .preDOSPHYSICALDISK,
TYPE        = (PRE,API),
GROUP       = FS,
DESC        = "(OS) DosPhysicalDisk Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
<LOG_CX>,
FMT         = "Function = <FMT_AX> Data Len = <FMT_BX>",
FMT         = "Parm Len = <FMT_CX>",

```

***DT postDOSPHYSICALDISK - DosPhysicalDisk post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSPHYSICALDISK>,
TP          = .postDOSPHYSICALDISK,
TYPE        = (POST,API),
GROUP       = FS,
DESC        = "(OS) DosPhysicalDisk Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"

```

***DT preDOSREALLOCHUGE - DosReallocHuge pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSREALLOCHUGE>,
TP          = .preDOSREALLOCHUGE,
TYPE        = (PRE,API),
GROUP       = SEL,
DESC        = "(OS) DosReallocHuge Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
<LOG_CX>,
FMT         = "New size = <FMT_AX><FMT_BX> Selector = <FMT_CX>"

```

***DT postDOSREALLOCHUGE - DosReallocHuge post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSREALLOCHUGE>,
TP      = .postDOSREALLOCHUGE,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosReallocHuge Post-Invocation",
<LOG_RETVAL>,
FMT      = "Return code = <FMT_RETVAL>"

```

*****DT preDOSREALLOCSEG - DosReallocSeg pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSREALLOCSEG>,
TP      = .preDOSREALLOCSEG,
TYPE    = (PRE,API),
GROUP   = SEL,
DESC    = "(OS) DosReallocSeg Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT      = "Selector = <FMT_AX>  New size = <FMT_BX>"

```

*****DT postDOSREALLOCSEG - DosAllocSeg post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSREALLOCSEG>,
TP      = .postDOSREALLOCSEG,
TYPE    = (POST,API),
GROUP   = SEL,
DESC    = "(OS) DosReallocSeg Post-Invocation",
<LOG_RETVAL>,
FMT      = "Return code = <FMT_RETVAL>"

```

*****DT preDOSSEMSETWAIT - DosSemSetWait pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSSEMSETWAIT>,
TP      = .preDOSSEMSETWAIT,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosSemSetWait Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT      = "Handle = <FMT_AX><FMT_BX>"

```

*****DT postDOSSEMSETWAIT - DosSemSetWait post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSSEMSETWAIT>,
TP      = .postDOSSEMSETWAIT,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) DosSemSetWait Post-Invocation",
<LOG_RETVAL>,
FMT      = "Return code = <FMT_RETVAL>"

```

*****DT preDOSSEND SIGNAL - DosSendSignal pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSSEND SIGNAL>,
TP      = .preDOSSEND SIGNAL,
TYPE    = (PRE,API),
GROUP   = SIG,
DESC    = "(OS) DosSendSignal Pre-Invocation",
<LOG_AX>,
<LOG_CX>,
FMT      = "Signal number = <FMT_AX>  CSID = <FMT_CX>"

```

*****DT postDOSSEND SIGNAL - DosSendSignal post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSSEENDSIGNAL>,
TP      = .postDOSSEENDSIGNAL,
TYPE    = (POST,API),
GROUP   = SIG,
DESC    = "(OS) DosSendSignal Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSSETSIGHANDLER - DosSetSigHandler pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSSETSIGHANDLER>,
TP      = .preDOSSETSIGHANDLER,
TYPE    = (PRE,API),
GROUP   = SIG,
DESC    = "(OS) DosSetSigHandler Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT     = "Signal number = <FMT_AX> Action = <FMT_BX>"

```

***DT postDOSSETSIGHANDLER - DosSetSigHandler post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSSETSIGHANDLER>,
TP      = .postDOSSETSIGHANDLER,
TYPE    = (POST,API),
GROUP   = SIG,
DESC    = "(OS) DosSetSigHandler Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSSETVEC - DosSetVec pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSSETVEC>,
TP      = .preDOSSETVEC,
TYPE    = (PRE,API),
GROUP   = SIG,
DESC    = "(OS) DosSetVec Pre-Invocation",
<LOG_CX>,
<LOG_BX>,
<LOG_AX>,
FMT     = "Return address = <FMT_CX><FMT_BX> Vector = <FMT_AX>"

```

***DT postDOSSETVEC - DosSetVec post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSSETVEC>,
TP      = .postDOSSETVEC,
TYPE    = (POST,API),
GROUP   = SIG,
DESC    = "(OS) DosSetVec Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSSYSTEMSERVICE - DosSystemService pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSSYSTEMSERVICE>,
TP      = .preDOSSYSTEMSERVICE,
TYPE    = (PRE, API),
GROUP   = TK,
DESC    = "(OS) DosSystemService Pre-Invocation",
<LOG_DI>,
FMT     = "Function = <FMT_DI>"

```

***DT postDOSSYSTEMSERVICE - DosSystemService post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSYSTEMSERVICE>,
TP      = .postDOSSYSTEMSERVICE,
TYPE    = (POST, API),
GROUP   = TK,
DESC    = "(OS) DosSystemService Post-Invocation",
<LOG_RETVAL>
FMT      = "Return Code = <FMT_RETVAL>"
```

Task operations (API 1/2)

***DT preDOSSETDATETIME - DosSetDateTime pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSSETDATETIME>,
TP      = .preDOSSETDATETIME,
TYPE    = (PRE, API),
GROUP   = TIM,
DESC    = "(OS) DosSetDateTime Pre-Invocation",
<LOG_BX>,
<LOG_SI>,
FMT      = "DateTime structure at <FMT_BX><FMT_SI>"
```

***DT postDOSSETDATETIME - DosSetDateTime post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSETDATETIME>,
TP      = .postDOSSETDATETIME,
TYPE    = (POST, API),
GROUP   = TIM,
DESC    = "(OS) DosSetDateTime Post-Invocation",
<LOG_RETVAL>,
FMT      = "Return code = <FMT_RETVAL>"
```

***DT preDOSGETDATETIME - DosGetDateTime pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSGETDATETIME>,
TP      = .preDOSGETDATETIME,
TYPE    = (PRE, API),
GROUP   = TIM,
DESC    = "(OS) DosGetDateTime Pre-Invocation",
<LOG_BX>,
<LOG_DI>,
FMT      = "DateTime structure at <FMT_BX><FMT_DI>"
```

***DT postDOSGETDATETIME - DosGetDateTime post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSGETDATETIME>,
TP      = .postDOSGETDATETIME,
TYPE    = (POST, API),
GROUP   = TIM,
DESC    = "(OS) DosGetDateTime Post-Invocation",
<LOG_RETVAL>,
FMT      = "Return code = <FMT_RETVAL>"
```

***DT preDOSSETPRTY - DosSetPrtY pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSSETPRTY>,
TP      = .preDOSSETPRTY,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosSetPrty Pre-Invocation",
<LOG_DX>,
<LOG_BX>,
<LOG_AX>,
FMT     = "Scope = <FMT_DX>  Priority Class, Delta = <FMT_BX>",
FMT     = "ID = <FMT_AX>"

```

***DT postDOSSETPRTY - DosSetPrty post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSSETPRTY>,
TP      = .postDOSSETPRTY,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) DosSetPrty Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSGETPRTY - DosGetPrty pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSGETPRTY>,
TP      = .preDOSGETPRTY,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosGetPrty Pre-Invocation",
<LOG_AX>,
<LOG_DX>,
FMT     = "ID = <FMT_AX>  Scope = <FMT_DX>"

```

***DT postDOSGETPRTY - DosGetPrty post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSGETPRTY>,
TP      = .postDOSGETPRTY,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) DosGetPrty Post-Invocation",
<LOG_BX>,
<LOG_RETVAL>,
FMT     = "Priority = <FMT_BX>  Return code = <FMT_RETVAL>"

```

***DT preDOSBEEP - DosBeep pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSBEEP>,
TP      = .preDOSBEEP,
TYPE    = (PRE,API),
GROUP   = IO,
DESC    = "(OS) DosBeep Pre-Invocation",

```

***DT postDOSBEEP - DosBeep post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSBEEP>,
TP      = .postDOSBEEP,
TYPE    = (POST,API),
GROUP   = IO,
DESC    = "(OS) DosBeep Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSSETTRACEINFO - DosSetTraceInfo pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSSETTRACEINFO>,
TP      = .preDOSSETTRACEINFO,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosSetTraceInfo Pre-Invocation",
<LOG_BX>,
<LOG_CX>,
<LOG_DX>,
FMT     = "Request Code = <FMT_BX>  PID = <FMT_CX>",
FMT     = "Count = <FMT_DX>"

```

*****DT postDOSSETTRACEINFO - DosSetTraceInfo post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSSETTRACEINFO>,
TP      = .postDOSSETTRACEINFO,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) DosSetTraceInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

*****DT preDOSISETCP - DosISetCP pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSISETCP>,
TP      = .preDOSISETCP,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) DosISetCP Pre-Invocation",
<LOG_BX>,
FMT     = "Codepage ID = <FMT_BX>"

```

*****DT postDOSISETCP - DosISetCP post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSISETCP>,
TP      = .postDOSISETCP,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) DosISetCP Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

*****DT preDOSGETCP - DosGetCP pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSGETCP>,
TP      = .preDOSGETCP,
TYPE    = (PRE,API),
GROUP   = NLS,
DESC    = "(OS) DosGetCP Pre-Invocation",

```

*****DT postDOSGETCP - DosGetCP post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOSGETCP>,
TP      = .postDOSGETCP,
TYPE    = (POST,API),
GROUP   = NLS,
DESC    = "(OS) DosGetCP Post-Invocation",

```

*****DT preDOSCLIACCESS - DosCliAccess pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSCLIACCESS>,
TP          = .preDOSCLIACCESS,
TYPE        = (PRE,API),
GROUP       = TK,
DESC        = "(OS) DosCliAccess Pre-Invocation",
```

*****DT postDOSCLIACCESS - DosCliAccess post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSCLIACCESS>,
TP          = .postDOSCLIACCESS,
TYPE        = (POST,API),
GROUP       = TK,
DESC        = "(OS) DosCliAccess Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSPORTACCESS - DosPortAccess pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSPORTACCESS>,
TP          = .preDOSPORTACCESS,
TYPE        = (PRE,API),
GROUP       = TK,
DESC        = "(OS) DosPortAccess Pre-Invocation",
<LOG_DX>,
<LOG_CX>,
FMT         = "Last Port = <FMT_DX>  First Port = <FMT_CX>"
```

*****DT postDOSPORTACCESS - DosPortAccess post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPORTACCESS>,
TP          = .postDOSPORTACCESS,
TYPE        = (POST,API),
GROUP       = TK,
DESC        = "(OS) DosPortAccess Post-Invocation",
<LOG_RETVAL>,
FMT         = "Return code = <FMT_RETVAL>"
```

*****DT preDOSEENTERCRITSEC - DosEnterCritSec pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSEENTERCRITSEC>,
TP          = .preDOSEENTERCRITSEC,
TYPE        = (PRE,API),
GROUP       = TK,
DESC        = "(OS) DosEnterCritSec Pre-Invocation"
```

*****DT post2DOSEENTERCRITSEC - DosEnterCritSec post-invocation trace point**

```
TRACE MINOR = <RAS_POST2_DOSEENTERCRITSEC>,
TP          = .post2DOSEENTERCRITSEC,
TYPE        = (POST,API),
GROUP       = TK,
DESC        = "(OS) DosEnterCritSec Post-Invocation",
REGS        = (AX),
FMT         = "Return code = <FMT_WORD>"
```

*****DT preDOSEXITCRITSEC - DosExitCritSec pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSEXITCRITSEC>,
TP          = .preDOSEXITCRITSEC,
TYPE        = (PRE,API),
GROUP       = TK,
```

```
DESC = "(OS) DosExitCritSec Pre-Invocation"
```

*****DT post2DOSEXITCRITSEC - DosExitCritSec post-invocation trace point**

```
TRACE MINOR = <RAS_POST2_DOSEXITCRITSEC>,
TP          = .post2DOSEXITCRITSEC,
TYPE        = (POST,API),
GROUP       = TK,
DESC        = "(OS) DosExitCritSec Post-Invocation",
REGS        = (AX),
FMT         = "Return code = <FMT_WORD>"
```

*****DT preDOS32ALIASMEM - Dos32AliasMem pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32ALIASMEM>,
TP          = .preDOS32ALIASMEM,
TYPE        = (PRE,API),
GROUP       = VM,
DESC        = "(OS) Dos32AliasMem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT         = "Address = <FMT_C_PTR> Size = <FMT_C_DWORD>"
```

*****DT postDOS32ALIASMEM - Dos32AliasMem post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32ALIASMEM>,
TP          = .postDOS32ALIASMEM,
TYPE        = (POST,API),
GROUP       = VM,
DESC        = "(OS) Dos32AliasMem Post-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM3>,IF,4),
MEM         = <LOG_C_RETVAL>,
FMT         = "Alias address = <FMT_C_PTR> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32ALLOCMEM - Dos32AllocMem pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32ALLOCMEM>,
TP          = .preDOS32ALLOCMEM,
TYPE        = (PRE,API),
GROUP       = VM,
DESC        = "(OS) Dos32AllocMem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM2>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM3>,D,4),
FMT         = "Address = <FMT_C_PTR> Size = <FMT_C_DWORD>",
FMT         = "Flags = <FMT_C_DWORD>"
```

*****DT postDOS32ALLOCMEM - Dos32AllocMem post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32ALLOCMEM>,
TP          = .postDOS32ALLOCMEM,
TYPE        = (POST,API),
GROUP       = VM,
DESC        = "(OS) Dos32AllocMem Post-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,IF,4),
MEM         = <LOG_C_RETVAL>,
FMT         = "Address = <FMT_C_PTR> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32ALLOCPROTECTEDMEM - Dos32AllocProtectedMem pre-invocation tp.**

```

TRACE MINOR = <RAS_PRE_DOS32ALLOCPROTECTEDMEM>,
TP          = .preDOS32ALLOCPROTECTEDMEM,
TYPE       = (PRE,API),
GROUP      = VM,
DESC       = "(OS) Dos32AllocProtectedMem Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM        = (<KSTK32_ARGS>+<C_PARM4>,D,4),
FMT        = "Size = <FMT_C_DWORD>  Flags = <FMT_C_DWORD>"

```

*****DT postDOS32ALLOCPROTECTEDMEM - Dos32AllocProtectedMem post-invocation tp.**

```

TRACE MINOR = <RAS_POST_DOS32ALLOCPROTECTEDMEM>,
TP          = .postDOS32ALLOCPROTECTEDMEM,
TYPE       = (POST,API),
GROUP      = VM,
DESC       = "(OS) Dos32AllocProtectedMem Post-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>,IF,4),
<LOG_C_RETVAL>,
FMT        = "Address = <FMT_C_PTR>  Return code = <FMT_C_RETVAL>"

```

*****DT preDOS32ALLOCSHAREDMEM - Dos32AllocSharedMem pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOS32ALLOCSHAREDMEM>,
TP          = .preDOS32ALLOCSHAREDMEM,
TYPE       = (PRE,API),
GROUP      = VM,
DESC       = "(OS) Dos32AllocSharedMem Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM        = (<KSTK32_ARGS>+<C_PARM4>,D,4),
FMT        = "Size = <FMT_C_DWORD>  Flags = <FMT_C_DWORD>"

```

*****DT postDOS32ALLOCSHAREDMEM - Dos32AllocSharedMem post-invocation tp.**

```

TRACE MINOR = <RAS_POST_DOS32ALLOCSHAREDMEM>,
TP          = .postDOS32ALLOCSHAREDMEM,
TYPE       = (POST,API),
GROUP      = VM,
DESC       = "(OS) Dos32AllocSharedMem Post-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>,IF,4),
<LOG_C_RETVAL>,
FMT        = "Address = <FMT_C_PTR>  Return code = <FMT_C_RETVAL>"

```

*****DT preDOS32CREATETHREAD - Dos32CreateThread pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOS32CREATETHREAD>,
TP          = .preDOS32CREATETHREAD,
TYPE       = (PRE,API),
GROUP      = TK,
DESC       = "(OS) Dos32CreateThread Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM5>,D,4), /* cbstack */
MEM        = (<KSTK32_ARGS>+<C_PARM4>,D,4), /* fl */
MEM        = (<KSTK32_ARGS>+<C_PARM3>,D,4), /* parg */
MEM        = (<KSTK32_ARGS>+<C_PARM2>,D,4), /* pfnstart */
FMT        = "Stack size = <FMT_C_DWORD>  Flags = <FMT_C_DWORD>",
FMT        = "Arg pointer = <FMT_C_PTR>  Starting EIP = <FMT_C_PTR>"

```

*****DT postDOS32CREATETHREAD - Dos32CreateThread post-invocation trace point**

```

TRACE MINOR = <RAS_POST_DOS32CREATETHREAD>,
TP          = .postDOS32CREATETHREAD,

```

```

TYPE = (POST,API),
GROUP = TK,
DESC = "(OS) Dos32CreateThread Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,IF,4), /* *pultid */
<LOG_C_RETVAL>, /* return code */
FMT = "Thread ID = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32KILLTHREAD - Dos32KillThread pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32KILLTHREAD>,
TP = .preDOS32KILLTHREAD,
TYPE = (PRE,API),
GROUP = TK,
DESC = "(OS) Dos32KillThread Pre-Invocation",
MEM = (<KSTK32_ARGS_NT>+<C_PARM1>,D,4),
FMT = "Tid = <FMT_C_DWORD>"

```

***DT postDOS32KILLTHREAD - Dos32KillThread post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32KILLTHREAD>,
TP = .postDOS32KILLTHREAD,
TYPE = (POST,API),
GROUP = TK,
DESC = "(OS) Dos32KillThread Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOSCREATETHREAD - DosCreateThread pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCREATETHREAD>,
TP = .preDOSCREATETHREAD,
TYPE = (PRE,API),
GROUP = TK,
DESC = "(OS) DosCreateThread Pre-Invocation",
MEM = (<KSTK32_ARGS_NT>+<C_PARM3>,D,4), /* start address */
MEM = (<KSTK32_ARGS_NT>+<C_PARM1>,D,4), /* stack address */
FMT = "Transfer Address = <FMT_C_P16> Stack Address = <FMT_C_P16>"

```

***DT postDOSCREATETHREAD - DosCreateThread post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCREATETHREAD>,
TP = .postDOSCREATETHREAD,
TYPE = (POST,API),
GROUP = TK,
DESC = "(OS) DosCreateThread Post-Invocation",
MEM = (<KSTK32_ARGS_NT>+<C_PARM2>,I,2), /* thread i.d. */
<LOG_C_RETVAL>,
FMT = "Thread ID = <FMT_C_WORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32DEBUG - Dos32Debug pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32DEBUG>,
TP = .preDOS32DEBUG,
TYPE = (PRE,API),
GROUP = TK,
DESC = "(OS) Dos32Debug Dos Pre-Invocation"

```

***DT postDOS32DEBUG - Dos32Debug post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32DEBUG>,
TP          = .postDOS32DEBUG,
TYPE       = (POST, API),
GROUP      = TK,
DESC       = "(OS) Dos32Debug Dos Post-Invocation",
<LOG_C_RETVAL>,
FMT        = "Return code = <FMT_C_RETVAL>"
```

***DT preDOSEXIT - DosExit pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSEXIT>,
TP          = .preDOSEXIT,
TYPE       = (PRE, API),
GROUP      = TK,
DESC       = "(OS) DosExit Dos Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM2>,D,2),
MEM        = (<KSTK32_ARGS>+<C_PARM1>,D,2),
FMT        = "Function = <FMT_C_WORD>   ResultCode = <FMT_C_WORD>"
```

***DT postDOSEXIT - DosExit post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSEXIT>,
TP          = .postDOSEXIT,
TYPE       = (POST, API),
GROUP      = TK,
DESC       = "(OS) Dos32Exit Dos Post-Invocation",
<LOG_C_RETVAL>,
FMT        = "Return code = <FMT_C_RETVAL>"
```

***DT preDOSEXITLIST - DosExitList pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSEXITLIST>,
TP          = .preDOSEXITLIST,
TYPE       = (PRE, API),
GROUP      = TK,
DESC       = "(OS) DosExitList Dos Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM2>,D,2),
MEM        = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT        = "Function = <FMT_C_WORD>   Address = <FMT_C_PTR>"
```

***DT postDOSEXITLIST - DosExitList post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSEXITLIST>,
TP          = .postDOSEXITLIST,
TYPE       = (POST, API),
GROUP      = TK,
DESC       = "(OS) Dos32ExitList Dos Post-Invocation",
<LOG_C_RETVAL>,
FMT        = "Return code = <FMT_C_RETVAL>"
```

***DT preDOS32EXITLIST - Dos32ExitList pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOS32EXITLIST>,
TP          = .preDOS32EXITLIST,
TYPE       = (PRE, API),
GROUP      = TK,
DESC       = "(OS) Dos32ExitList Dos Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM        = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT        = "Function = <FMT_C_DWORD>   Address = <FMT_C_PTR>"
```

***DT postDOS32EXITLIST - Dos32ExitList post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32EXITLIST>,
TP          = .postDOS32EXITLIST,
TYPE        = (POST, API),
GROUP       = TK,
DESC        = "(OS) Dos32ExitList Dos Post-Invocation",
<LOG_C_RETVAL>,
FMT         = "Return code = <FMT_C_RETVAL>"
```

***DT preDOS32FREEMEM - Dos32FreeMem pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOS32FREEMEM>,
TP          = .preDOS32FREEMEM,
TYPE        = (PRE, API),
GROUP       = VM,
DESC        = "(OS) Dos32FreeMem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
FMT         = "Address = <FMT_C_PTR>"
```

***DT postDOS32FREEMEM - Dos32FreeMem post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32FREEMEM>,
TP          = .postDOS32FREEMEM,
TYPE        = (POST, API),
GROUP       = VM,
DESC        = "(OS) Dos32FreeMem Post-Invocation",
<LOG_C_RETVAL>,
FMT         = "Return code = <FMT_C_RETVAL>"
```

***DT preDOSFREEMODULE - DosFreeModule pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSFREEMODULE>,
TP          = .preDOSFREEMODULE,
TYPE        = (PRE, API),
GROUP       = LDR,
DESC        = "(OS) DosFreeModule Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
FMT         = "Module Handle = <FMT_C_PTR>"
```

***DT postDOSFREEMODULE - DosFreeModule post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSFREEMODULE>,
TP          = .postDOSFREEMODULE,
TYPE        = (POST, API),
GROUP       = LDR,
DESC        = "(OS) DosFreeModule Post-Invocation",
<LOG_C_RETVAL>,
FMT         = "Return code = <FMT_C_RETVAL>"
```

***DT preDOS32FREERESOURCE - Dos32FreeResource pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOS32FREERESOURCE>,
TP          = .preDOS32FREERESOURCE,
TYPE        = (PRE, API),
GROUP       = LDR,
DESC        = "(OS) Dos32FreeResource Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
FMT         = "Pointer = <FMT_C_PTR>"
```

***DT postDOS32FREERESOURCE - Dos32FreeResource post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32FREERESOURCE>,
TP      = .postDOS32FREERESOURCE,
TYPE    = (POST,API),
GROUP   = LDR,
DESC    = "(OS) Dos32FreeResource Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32GETNAMEDSHAREDMMEM - Dos32GetNamedSharedMem pre-invocation tp.**

```
TRACE MINOR = <RAS_PRE_DOS32GETNAMEDSHAREDMMEM>,
TP      = .preDOS32GETNAMEDSHAREDMMEM,
TYPE    = (PRE,API),
GROUP   = VM,
DESC    = "(OS) Dos32GetNamedSharedMem Pre-Invocation",
ASCIIIZ = (<KSTK32_ARGS>+<C_PARM2>, IF, <MAXPATHLEN>),
MEM     = (<KSTK32_ARGS>+<C_PARM3>, D, 4),
FMT     = "Name = <FMT_C_ASCIIIZ>",
FMT     = "Flags = <FMT_C_DWORD>"
```

*****DT postDOS32GETNAMEDSHAREDMMEM - Dos32GetNamedSharedMem post-invocation tp.**

```
TRACE MINOR = <RAS_POST_DOS32GETNAMEDSHAREDMMEM>,
TP      = .postDOS32GETNAMEDSHAREDMMEM,
TYPE    = (POST,API),
GROUP   = VM,
DESC    = "(OS) Dos32GetNamedSharedMem Post-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>, IF, 4),
<LOG_C_RETVAL>,
FMT     = "Address = <FMT_C_PTR> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32QUERYPROCADDR - Dos32QueryProcAddr pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYPROCADDR>,
TP      = .preDOS32QUERYPROCADDR,
TYPE    = (PRE,API),
GROUP   = LDR,
DESC    = "(OS) Dos32QueryProcAddr Pre-Invocation",
ASCIIIZ = (<KSTK32_ARGS>+<C_PARM3>, IF, <MAXPATHLEN>),
MEM     = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>, D, 4),
FMT     = "Name = <FMT_C_ASCIIIZ>",
FMT     = "hMod = <FMT_C_DWORD> Ord = <FMT_C_DWORD>"
```

*****DT postDOS32QUERYPROCADDR - Dos32QueryProcAddr post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32QUERYPROCADDR>,
TP      = .postDOS32QUERYPROCADDR,
TYPE    = (POST,API),
GROUP   = LDR,
DESC    = "(OS) Dos32QueryProcAddr Post-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM4>, IF, 4),
<LOG_C_RETVAL>,
FMT     = "Proc Addr = <FMT_C_PTR> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32GETRESOURCE - Dos32GetResource pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32GETRESOURCE>,
TP      = .preDOS32GETRESOURCE,
```

```

TYPE = (PRE,API),
GROUP = LDR,
DESC = "(OS) Dos32GetResource Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "hMod = <FMT_C_DWORD>  TypeID = <FMT_C_DWORD>",
FMT = "NameID = <FMT_C_DWORD>"

```

***DT postDOS32GETRESOURCE - Dos32GetResource post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32GETRESOURCE>,
TP = .postDOS32GETRESOURCE,
TYPE = (POST,API),
GROUP = LDR,
DESC = "(OS) Dos32GetResource Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM4>,IF,4),
<LOG_C_RETVAL>,
FMT = "Address = <FMT_C_PTR>   Return code = <FMT_C_RETVAL>"

```

***DT preDOS32QUERYRESOURCESIZE - Dos32QueryResourceSize pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32QUERYRESOURCESIZE>,
TP = .preDOS32QUERYRESOURCESIZE,
TYPE = (PRE,API),
GROUP = LDR,
DESC = "(OS) Dos32QueryResource Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "hMod = <FMT_C_DWORD>  TypeID = <FMT_C_DWORD>",
FMT = "NameID = <FMT_C_DWORD>"

```

***DT postDOS32QUERYRESOURCESIZE - Dos32QueryResourceSize post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32QUERYRESOURCESIZE>,
TP = .postDOS32QUERYRESOURCESIZE,
TYPE = (POST,API),
GROUP = LDR,
DESC = "(OS) Dos32QueryResourceSize Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM4>,IF,4),
<LOG_C_RETVAL>,
FMT = "Address = <FMT_C_PTR>   Return code = <FMT_C_RETVAL>"

```

***DT preDOS32INITIALIZEPORTHOLE - Dos32INITIALIZEPORTHOLE pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32INITIALIZEPORTHOLE>,
TP = .preDOS32INITIALIZEPORTHOLE,
TYPE = (PRE,API),
GROUP = LDR,
DESC = "(OS) Dos32InitializePorthole Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "Init Routine = <FMT_C_DWORD> entry type = <FMT_C_DWORD>"

```

***DT postDOS32INITIALIZEPORTHOLE - Dos32INITIALIZEPORTHOLE post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32INITIALIZEPORTHOLE>,
TP = .postDOS32INITIALIZEPORTHOLE,
TYPE = (POST,API),

```

```
GROUP = LDR,  
DESC = "(OS) Dos32InitializePorthole Post-Invocation",  
<LOG_C_RETURN>,  
FMT = "Return code = <FMT_C_RETURN>"
```

*****DT preDOS32QUERYHEADERINFO - Dos32QueryHeaderInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYHEADERINFO>,  
TP = .preDOS32QUERYHEADERINFO,  
TYPE = (PRE,API),  
GROUP = LDR,  
DESC = "(OS) Dos32QueryHeaderInfo Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM4>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM5>,D,4),  
FMT = "hMod = <FMT_C_DWORD> index = <FMT_C_DWORD>",  
FMT = "phdr = <FMT_C_DWORD> cb = <FMT_C_DWORD>",  
FMT = "subfunc = <FMT_C_DWORD>"
```

*****DT postDOS32QUERYHEADERINFO - Dos32QueryHeaderInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32QUERYHEADERINFO>,  
TP = .postDOS32QUERYHEADERINFO,  
TYPE = (POST,API),  
GROUP = LDR,  
DESC = "(OS) Dos32QueryHeaderInfo Post-Invocation",  
<LOG_C_RETURN>,  
FMT = "Return code = <FMT_C_RETURN>"
```

*****DT preDOS32QUERYPROCTYPE - Dos32QueryProcType pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYPROCTYPE>,  
TP = .preDOS32QUERYPROCTYPE,  
TYPE = (PRE,API),  
GROUP = LDR,  
DESC = "(OS) Dos32QueryProcType Pre-Invocation",  
ASCIIIZ = (<KSTK32_ARGS>+<C_PARM3>,IF,<MAXPATHLEN>),  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "Name = <FMT_C_ASCIIIZ>",  
FMT = "hMod = <FMT_C_DWORD> Ord = <FMT_C_DWORD>"
```

*****DT postDOS32QUERYPROCTYPE - Dos32QueryProcType post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32QUERYPROCTYPE>,  
TP = .postDOS32QUERYPROCTYPE,  
TYPE = (POST,API),  
GROUP = LDR,  
DESC = "(OS) Dos32QueryProcType Post-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM4>,IF,4),  
<LOG_C_RETURN>,  
FMT = "Proc Type = <FMT_C_PTR> Return code = <FMT_C_RETURN>"
```

*****DT preDOS32GETSHAREDMEM - Dos32GetSharedMem pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32GETSHAREDMEM>,  
TP = .preDOS32GETSHAREDMEM,  
TYPE = (PRE,API),
```

```
GROUP = VM,
DESC = "(OS) Dos32GetSharedMem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "Address = <FMT_C_PTR> Flags = <FMT_C_DWORD>"
```

*****DT postDOS32GETSHAREDMEM - Dos32GetSharedMem post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32GETSHAREDMEM>,
TP = .postDOS32GETSHAREDMEM,
TYPE = (POST,API),
GROUP = VM,
DESC = "(OS) Dos32GetSharedMem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32GIVESHAREDMEM - Dos32GiveSharedMem pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32GIVESHAREDMEM>,
TP = .preDOS32GIVESHAREDMEM,
TYPE = (PRE,API),
GROUP = VM,
DESC = "(OS) Dos32GiveSharedMem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
FMT = "Address = <FMT_C_PTR> Process ID = <FMT_C_DWORD>",
FMT = "Flags = <FMT_C_DWORD>"
```

*****DT postDOS32GIVESHAREDMEM - Dos32GiveSharedMem post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32GIVESHAREDMEM>,
TP = .postDOS32GIVESHAREDMEM,
TYPE = (POST,API),
GROUP = VM,
DESC = "(OS) Dos32GiveSharedMem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32STARTTIMER - Dos32StartTimer pre-invocation trace point**

```
TRACE MINOR = 379,
TP = .preDOS32STARTTIMER,
TYPE = (PRE,API),
GROUP = TIM,
DESC = "(OS) Dos32StartTimer Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "Interval = <FMT_C_DWORD>",
FMT = "Semaphore at <FMT_C_DWORD>"
```

*****DT postDOS32STARTTIMER - Dos32StartTimer post-invocation trace point**

```
TRACE MINOR = 380,
TP = .postDOS32STARTTIMER,
TYPE = (POST,API),
GROUP = TIM,
DESC = "(OS) Dos32StartTimer Post-Invocation",
<LOG_C_RETVAL>,
MEM = (<KSTK32_ARGS>+<C_PARM3>,IF,2),
FMT = "Return code = <FMT_C_RETVAL> Handle = <FMT_C_WORD>"
```

*****DT preDOS32STOPTIMER - Dos32StopTimer pre-invocation trace point**

```
TRACE MINOR = 312,
TP      = .preDOS32STOPTIMER,
TYPE    = (PRE,API),
GROUP   = TIM,
DESC    = "(OS) Dos32StopTimer Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,2),
FMT     = "Handle = <FMT_C_WORD>"
```

*****DT postDOS32STOPTIMER - Dos32StopTimer post-invocation trace point**

```
TRACE MINOR = 155,
TP      = .postDOS32STOPTIMER,
TYPE    = (POST,API),
GROUP   = TIM,
DESC    = "(OS) Dos32StopTimer Post-Invocation",
        <LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32ASYNCTIMER - Dos32AsyncTimer pre-invocation trace point**

```
TRACE MINOR = 377,
TP      = .preDOS32ASYNCTIMER,
TYPE    = (PRE,API),
GROUP   = TIM,
DESC    = "(OS) Dos32AsyncTimer Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
FMT     = "Interval = <FMT_C_DWORD>",
FMT     = "Semaphore at <FMT_C_DWORD>"
```

*****DT postDOS32ASYNCTIMER - Dos32AsyncTimer post-invocation trace point**

```
TRACE MINOR = 378,
TP      = .postDOS32ASYNCTIMER,
TYPE    = (POST,API),
GROUP   = TIM,
DESC    = "(OS) Dos32AsyncTimer Post-Invocation",
        <LOG_C_RETVAL>,
MEM     = (<KSTK32_ARGS>+<C_PARM3>,IF,2),
FMT     = "Return code = <FMT_C_RETVAL> Handle = <FMT_C_WORD>"
```

*****DT preDOS32ISTARTTIMER - Dos32StartTimer pre-invocation trace point**

```
TRACE MINOR = 311,
TP      = .preDOS32ISTARTTIMER,
TYPE    = (PRE,API),
GROUP   = TIM,
DESC    = "(OS) DosTimerStart Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT     = "Interval = <FMT_C_DWORD>",
FMT     = "Semaphore at <FMT_C_DWORD>"
```

*****DT postDOS32ISTARTTIMER - Dos32StartTimer post-invocation trace point**

```
TRACE MINOR = 154,
TP      = .postDOS32ISTARTTIMER,
TYPE    = (POST,API),
GROUP   = TIM,
DESC    = "(OS) DosTimerStart Post-Invocation",
```

```

REGS = (AX),
MEM = (<KSTK32_ARGS>+<C_PARM3>, IF, 2),
FMT = "Return code = %W Handle = <FMT_C_WORD>"

```

***DT preDOS32IASYNCTIMER - Dos32AsyncTimer pre-invocation trace point

```

TRACE MINOR = 310,
TP = .preDOS32IASYNCTIMER,
TYPE = (PRE, API),
GROUP = TIM,
DESC = "(OS) DosTimerAsync Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
MEM = (<KSTK32_ARGS>+<C_PARM2>, D, 4),
FMT = "Interval = <FMT_C_DWORD>",
FMT = "Semaphore at <FMT_C_DWORD>"

```

***DT postDOS32IASYNCTIMER - Dos32AsyncTimer post-invocation trace point

```

TRACE MINOR = 153,
TP = .postDOS32IASYNCTIMER,
TYPE = (POST, API),
GROUP = TIM,
DESC = "(OS) DosTimerAsync Post-Invocation",
REGS = (AX),
MEM = (<KSTK32_ARGS>+<C_PARM3>, IF, 2),
FMT = "Return code = %W Handle = <FMT_C_WORD>"

```

Task operations (API 2/2)

***DT preDOS32QUERYMEM - Dos32QueryMem pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32QUERYMEM>,
TP = .preDOS32QUERYMEM,
TYPE = (PRE, API),
GROUP = VM,
DESC = "(OS) Dos32QueryMem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
MEM = (<KSTK32_ARGS>+<C_PARM2>, IF, 4),
FMT = "Address = <FMT_C_PTR> Size = <FMT_C_DWORD>"

```

***DT postDOS32QUERYMEM - Dos32QueryMem post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32QUERYMEM>,
TP = .postDOS32QUERYMEM,
TYPE = (POST, API),
GROUP = VM,
DESC = "(OS) Dos32QueryMem Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM2>, IF, 4),
MEM = (<KSTK32_ARGS>+<C_PARM3>, IF, 4),
<LOG_C_RETVAL>,
FMT = "Actual size = <FMT_C_DWORD> Flags = <FMT_C_DWORD>",
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32QUERYMEMSTATE - Dos32QueryMemState pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32QUERYMEMSTATE>,
TP          = .preDOS32QUERYMEMSTATE,
TYPE       = (PRE, API),
GROUP      = VM,
DESC       = "(OS) Dos32QueryMemState Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
MEM        = (<KSTK32_ARGS>+<C_PARM2>, IF, 4),
FMT        = "Address = <FMT_C_PTR> Size = <FMT_C_DWORD>"

```

***DT postDOS32QUERYMEMSTATE - Dos32QueryMemState post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32QUERYMEMSTATE>,
TP          = .postDOS32QUERYMEMSTATE,
TYPE       = (POST, API),
GROUP      = VM,
DESC       = "(OS) Dos32QueryMemState Post-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM2>, IF, 4),
MEM        = (<KSTK32_ARGS>+<C_PARM3>, IF, 4),
<LOG_C_RETVAL>,
FMT        = "Actual size = <FMT_C_DWORD> Flags = <FMT_C_DWORD>",
FMT        = "Return code = <FMT_C_RETVAL>"

```

***DT preDOSRESUMETHREAD - DosResumeThread pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSRESUMETHREAD>,
TP          = .preDOSRESUMETHREAD,
TYPE       = (PRE, API),
GROUP      = TK,
DESC       = "(OS) DosResumeThread Dos Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>, D, 2),
FMT        = "Function = <FMT_C_WORD>"

```

***DT postDOSRESUMETHREAD - DosResumeThread post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSRESUMETHREAD>,
TP          = .postDOSRESUMETHREAD,
TYPE       = (POST, API),
GROUP      = TK,
DESC       = "(OS) Dos32ResumeThread Dos Post-Invocation",
<LOG_C_RETVAL>,
FMT        = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32SETMEM - Dos32SetMem pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32SETMEM>,
TP          = .preDOS32SETMEM,
TYPE       = (PRE, API),
GROUP      = VM,
DESC       = "(OS) Dos32SetMem Pre-Invocation",
MEM        = (<KSTK32_ARGS>+<C_PARM1>, D, 4),
MEM        = (<KSTK32_ARGS>+<C_PARM2>, D, 4),
MEM        = (<KSTK32_ARGS>+<C_PARM3>, D, 4),
FMT        = "Address = <FMT_C_PTR> Size = <FMT_C_DWORD>",
FMT        = "Flags = <FMT_C_DWORD>"

```

***DT postDOS32SETMEM - Dos32SetMem post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32SETMEM>,
TP          = .postDOS32SETMEM,
TYPE       = (POST, API),
GROUP      = VM,
DESC       = "(OS) Dos32SetMem Post-Invocation",

```

```
<LOG_C_RETVAL>,  
FMT    = "Return code = <FMT_C_RETVAL>"
```

***DT preDOSSLEEP - DosSleep pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSSLEEP>,  
TP          = .preDOSSLEEP,  
TYPE        = (PRE, API),  
GROUP       = TK,  
DESC        = "(OS) DosSleep Pre-Invocation",  
MEM         = (<KSTK32_ARGS_NT>+<C_PARM1>,D,4),  
FMT         = "Timeout Interval = <FMT_C_DWORD>"
```

***DT postDOSSLEEP - DosSleep post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSLEEP>,  
TP          = .postDOSSLEEP,  
TYPE        = (POST, API),  
GROUP       = TK,  
DESC        = "(OS) DosSleep Post-Invocation",  
<LOG_C_RETVAL>,  
FMT         = "Return code = <FMT_C_RETVAL>"
```

***DT preDOSSUSPENDTHREAD - DosSuspendThread pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSSUSPENDTHREAD>,  
TP          = .preDOSSUSPENDTHREAD,  
TYPE        = (PRE, API),  
GROUP       = TK,  
DESC        = "(OS) DosSuspendThread Dos Pre-Invocation",  
MEM         = (<KSTK32_ARGS>+<C_PARM2>,D,2),  
FMT         = "Function = <FMT_C_WORD>"
```

***DT postDOSSUSPENDTHREAD - DosSuspendThread post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSSUSPENDTHREAD>,  
TP          = .postDOSSUSPENDTHREAD,  
TYPE        = (POST, API),  
GROUP       = TK,  
DESC        = "(OS) Dos32SuspendThread Dos Post-Invocation",  
<LOG_C_RETVAL>,  
FMT         = "Return code = <FMT_C_RETVAL>"
```

***DT preDOS32CREATEEVENTSEM - DOS32CREATEEVENTSEM pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOS32CREATEEVENTSEM>,  
TP          = .preDOS32CREATEEVENTSEM,  
TYPE        = (PRE, API),  
GROUP       = SEM,  
DESC        = "(OS) Dos32CreateEventSem Pre-Invocation",  
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM         = (<KSTK32_ARGS>+<C_PARM3>,D,4),  
MEM         = (<KSTK32_ARGS>+<C_PARM4>,D,4),  
FMT         = "Name ptr = <FMT_C_PTR>  Attribs = <FMT_C_DWORD>",  
FMT         = "Initial State = <FMT_C_DWORD>"
```

***DT postDOS32CREATEEVENTSEM - DOS32CREATEEVENTSEM post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32CREATEEVENTSEM>,  
TP          = .postDOS32CREATEEVENTSEM,
```

```

TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
<LOG_C_RETVAL>,
FMT = "Handle = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32OPENEVENTSEM - DOS32OPENEVENTSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32OPENEVENTSEM>,
TP = .preDOS32OPENEVENTSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32OpenEventSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "Name Ptr = <FMT_C_PTR> phev = <FMT_C_PTR>"

```

***DT postDOS32OPENEVENTSEM - DOS32OPENEVENTSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32OPENEVENTSEM>,
TP = .postDOS32OPENEVENTSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
<LOG_C_RETVAL>,
FMT = "Handle = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32CLOSEEVENTSEM - DOS32CLOSEEVENTSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32CLOSEEVENTSEM>,
TP = .preDOS32CLOSEEVENTSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32OpenEventSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT = "Handle = <FMT_C_DWORD>"

```

***DT postDOS32CLOSEEVENTSEM - DOS32CLOSEEVENTSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32CLOSEEVENTSEM>,
TP = .postDOS32CLOSEEVENTSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32CloseEventSem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32RESETEVENTSEM - DOS32RESETEVENTSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32RESETEVENTSEM>,
TP = .preDOS32RESETEVENTSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32ResetEventSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT = "Handle = <FMT_C_DWORD>"

```

*****DT postDOS32RESETEVENTSEM - DOS32RESETEVENTSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32RESETEVENTSEM>,
TP      = .postDOS32RESETEVENTSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32ResetEventSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32POSTEVENTSEM - DOS32POSTEVENTSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32POSTEVENTSEM>,
TP      = .preDOS32POSTEVENTSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32PostEventSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT     = "Handle = <FMT_C_DWORD>"
```

*****DT postDOS32POSTEVENTSEM - DOS32POSTEVENTSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32POSTEVENTSEM>,
TP      = .postDOS32POSTEVENTSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32PostEventSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32WAITEVENTSEM - DOS32WAITEVENTSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32WAITEVENTSEM>,
TP      = .preDOS32WAITEVENTSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32WaitEvenSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT     = "Handle = <FMT_C_DWORD> Timeout = <FMT_C_DWORD>"
```

*****DT postDOS32WAITEVENTSEM - DOS32WAITEVENTSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32WAITEVENTSEM>,
TP      = .postDOS32WAITEVENTSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32WaitEventSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32QUERYEVENTSEM - DOS32QUERYEVENTSEM pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOS32QUERYEVENTSEM>,
TP          = .preDOS32QUERYEVENTSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32QueryEventSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT         = "Handle = <FMT_C_DWORD>"

```

***DT postDOS32QUERYEVENTSEM - DOS32QUERYEVENTSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32QUERYEVENTSEM>,
TP          = .postDOS32QUERYEVENTSEM,
TYPE        = (POST,API),
GROUP       = SEM,
DESC        = "(OS) Dos32QueryEventSem Post-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
             <LOG_C_RETVAL>,
FMT         = "State = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32CREATEMUTEXSEM - DOS32CREATEMUTEXSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32CREATEMUTEXSEM>,
TP          = .preDOS32CREATEMUTEXSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32CreateMutexSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM4>,D,4),
FMT         = "Name ptr = <FMT_C_PTR> Attribs = <FMT_C_DWORD>",
FMT         = "Initial State = <FMT_C_DWORD>"

```

***DT postDOS32CREATEMUTEXSEM - DOS32CREATEMUTEXSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32CREATEMUTEXSEM>,
TP          = .postDOS32CREATEMUTEXSEM,
TYPE        = (POST,API),
GROUP       = SEM,
DESC        = "(OS) Dos32CreateMutexSem Post-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
             <LOG_C_RETVAL>,
FMT         = "Handle = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32OPENMUTEXSEM - DOS32OPENMUTEXSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32OPENMUTEXSEM>,
TP          = .preDOS32OPENMUTEXSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32OpenMutexSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
FMT         = "Name ptr = <FMT_C_PTR> Handle = <FMT_C_DWORD>"

```

***DT postDOS32OPENMUTEXSEM - DOS32OPENMUTEXSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32OPENMUTEXSEM>,
TP          = .postDOS32OPENMUTEXSEM,

```

```

TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32OpenMutexSem Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
<LOG_C_RETVAL>,
FMT = "Handle = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32CLOSEMUTEXSEM - DOS32CLOSEMUTEXSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32CLOSEMUTEXSEM>,
TP = .preDOS32CLOSEMUTEXSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32CloseMutexSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT = "Handle = <FMT_C_DWORD>"

```

***DT postDOS32CLOSEMUTEXSEM - DOS32CLOSEMUTEXSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32CLOSEMUTEXSEM>,
TP = .postDOS32CLOSEMUTEXSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32CloseMutexSem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32REQUESTMUTEXSEM - DOS32REQUESTMUTEXSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32REQUESTMUTEXSEM>,
TP = .preDOS32REQUESTMUTEXSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32RequestMutexSem Pre-Invocation",
MEM = (RSS+SP+16+<C_PARM1>,D,4),
MEM = (RSS+SP+16+<C_PARM2>,D,4),
FMT = "Handle = <FMT_C_DWORD> Timeout = <FMT_C_DWORD>"

```

***DT postDOS32REQUESTMUTEXSEM - DOS32REQUESTMUTEXSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32REQUESTMUTEXSEM>,
TP = .postDOS32REQUESTMUTEXSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32RequestMutexSem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32RELEASEMUTEXSEM - DOS32RELEASEMUTEXSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32RELEASEMUTEXSEM>,
TP = .preDOS32RELEASEMUTEXSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32ReleaseMutexSem Pre-Invocation",
MEM = (RSS+SP+16+<C_PARM1>,D,4),
FMT = "Handle = <FMT_C_DWORD>"

```

*****DT postDOS32RELEASEMUTEXSEM - DOS32RELEASEMUTEXSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32RELEASEMUTEXSEM>,
TP      = .postDOS32RELEASEMUTEXSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32ReleaseMutexSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32QUERYMUTEXSEM - DOS32QUERYMUTEXSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYMUTEXSEM>,
TP      = .preDOS32QUERYMUTEXSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32QueryMutexSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT     = "Handle = <FMT_C_DWORD>"
```

*****DT postDOS32QUERYMUTEXSEM - DOS32QUERYMUTEXSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32QUERYMUTEXSEM>,
TP      = .postDOS32QUERYMUTEXSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32QueryMutexSem Post-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
MEM     = (<KSTK32_ARGS>+<C_PARM3>,IF,4),
MEM     = (<KSTK32_ARGS>+<C_PARM4>,IF,4),
<LOG_C_RETVAL>,
FMT     = "PID Owner = <FMT_C_DWORD>  TID Owner = <FMT_C_DWORD>",
FMT     = "Count = <FMT_C_DWORD>  Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32CREATEMUXWAITSEM - DOS32CREATEMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32CREATEMUXWAITSEM>,
TP      = .preDOS32CREATEMUXWAITSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32CreateMuxWaitSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM4>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM5>,D,4),
FMT     = "Name ptr = <FMT_C_PTR>  cSemRec = <FMT_C_DWORD>",
FMT     = "SemRec ptr = <FMT_C_PTR>  Attribs = <FMT_C_DWORD>"
```

*****DT postDOS32CREATEMUXWAITSEM - DOS32CREATEMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32CREATEMUXWAITSEM>,
TP      = .postDOS32CREATEMUXWAITSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32CreateMuxWaitSem Post-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
<LOG_C_RETVAL>,
FMT     = "Handle = <FMT_C_DWORD>  Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32OPENMUXWAITSEM - DOS32OPENMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32OPENMUXWAITSEM>,
TP          = .preDOS32OPENMUXWAITSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32OpenMuxWaitSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT         = "Name Ptr = <FMT_C_PTR>  phmux = <FMT_C_PTR>"
```

*****DT postDOS32OPENMUXWAITSEM - DOS32OPENMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32OPENMUXWAITSEM>,
TP          = .postDOS32OPENMUXWAITSEM,
TYPE        = (POST,API),
GROUP       = SEM,
DESC        = "(OS) Dos32OpenMuxWaitSem Post-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
            <LOG_C_RETVAL>,
FMT         = "Handle = <FMT_C_DWORD>  Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32CLOSEMUXWAITSEM - DOS32CLOSEMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32CLOSEMUXWAITSEM>,
TP          = .preDOS32CLOSEMUXWAITSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32CloseMuxWaitSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT         = "Handle = <FMT_C_DWORD>"
```

*****DT postDOS32CLOSEMUXWAITSEM - DOS32CLOSEMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32CLOSEMUXWAITSEM>,
TP          = .postDOS32CLOSEMUXWAITSEM,
TYPE        = (POST,API),
GROUP       = SEM,
DESC        = "(OS) Dos32CloseMuxWaitSem Post-Invocation",
            <LOG_C_RETVAL>,
FMT         = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32WAITMUXWAITSEM - DOS32WAITMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32WAITMUXWAITSEM>,
TP          = .preDOS32WAITMUXWAITSEM,
TYPE        = (PRE,API),
GROUP       = SEM,
DESC        = "(OS) Dos32WaitMuxWaitSem Pre-Invocation",
MEM         = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM         = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT         = "Handle = <FMT_C_DWORD>  Timeout = <FMT_C_DWORD>"
```

*****DT postDOS32WAITMUXWAITSEM - DOS32WAITMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32WAITMUXWAITSEM>,
TP      = .postDOS32WAITMUXWAITSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32WaitMuxWaitSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32ADDMUXWAITSEM - DOS32ADDMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32ADDMUXWAITSEM>,
TP      = .preDOS32ADDMUXWAITSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32AddMuxWaitSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,IF,8),
FMT     = "Handle = <FMT_C_DWORD> SemRec = <FMT_C_QWORD>"
```

*****DT postDOS32ADDMUXWAITSEM - DOS32ADDMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32ADDMUXWAITSEM>,
TP      = .postDOS32ADDMUXWAITSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32AddMuxWaitSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32DELETEMUXWAITSEM - DOS32DELETEMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32DELETEMUXWAITSEM>,
TP      = .preDOS32DELETEMUXWAITSEM,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) Dos32DeleteMuxWaitSem Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT     = "Mux Handle = <FMT_C_DWORD> Sem Handle = <FMT_C_DWORD>"
```

*****DT postDOS32DELETEMUXWAITSEM - DOS32DELETEMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32DELETEMUXWAITSEM>,
TP      = .postDOS32DELETEMUXWAITSEM,
TYPE    = (POST,API),
GROUP   = SEM,
DESC    = "(OS) Dos32DeleteMuxWaitSem Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32QUERYMUXWAITSEM - DOS32QUERYMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYMUXWAITSEM>,
TP      = .preDOS32QUERYMUXWAITSEM,
TYPE    = (PRE,API),
```

```

GROUP = SEM,
DESC = "(OS) Dos32QueryMuxWaitSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT = "Handle = <FMT_C_DWORD>"

```

***DT postDOS32QUERYMUXWAITSEM - DOS32QUERYMUXWAITSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32QUERYMUXWAITSEM>,
TP = .postDOS32QUERYMUXWAITSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32QueryMuxWaitSem Post-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM2>,IF,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM4>,IF,4),
<LOG_C_RETVAL>,
FMT = "cSemRec = <FMT_C_DWORD> pSemRec = <FMT_C_PTR>",
FMT = "Attribs = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32PMPOSTEVENTSEM - DOS32PMPOSTEVENTSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32PMPOSTEVENTSEM>,
TP = .preDOS32PMPOSTEVENTSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32PMPostEventSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
FMT = "PMHandle = <FMT_C_DWORD> Handle = <FMT_C_DWORD>"

```

***DT postDOS32PMPOSTEVENTSEM - DOS32PMPOSTEVENTSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32PMPOSTEVENTSEM>,
TP = .postDOS32PMPOSTEVENTSEM,
TYPE = (POST,API),
GROUP = SEM,
DESC = "(OS) Dos32PMPostEventSem Post-Invocation",
<LOG_C_RETVAL>,
FMT = "Return code = <FMT_C_RETVAL>"

```

***DT preDOS32PMWAITEVENTSEM - DOS32PMWAITEVENTSEM pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32PMWAITEVENTSEM>,
TP = .preDOS32PMWAITEVENTSEM,
TYPE = (PRE,API),
GROUP = SEM,
DESC = "(OS) Dos32PMWaitEvenSem Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT = "PMHandle = <FMT_C_DWORD> Handle = <FMT_C_DWORD>"
FMT = "Timeout = <FMT_C_DWORD>"

```

***DT postDOS32PMWAITEVENTSEM - DOS32PMWAITEVENTSEM post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32PMWAITEVENTSEM>,
TP = .postDOS32PMWAITEVENTSEM,
TYPE = (POST,API),

```

```
GROUP = SEM,  
DESC = "(OS) Dos32PMWaitEventSem Post-Invocation",  
<LOG_C_RETVAL>,  
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32PMWAITMUXWAITSEM - DOS32PMWAITMUXWAITSEM pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32PMWAITMUXWAITSEM>,  
TP = .preDOS32PMWAITMUXWAITSEM,  
TYPE = (PRE,API),  
GROUP = SEM,  
DESC = "(OS) Dos32WaitMuxWaitSem Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "Handle = <FMT_C_DWORD> Handle = <FMT_C_DWORD>"  
FMT = "Timeout = <FMT_C_DWORD>"
```

*****DT postDOS32WAITMUXWAITSEM - DOS32WAITMUXWAITSEM post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32PMWAITMUXWAITSEM>,  
TP = .postDOS32PMWAITMUXWAITSEM,  
TYPE = (POST,API),  
GROUP = SEM,  
DESC = "(OS) Dos32PMWaitMuxWaitSem Post-Invocation",  
<LOG_C_RETVAL>,  
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32WAITTHREAD - DOS32WAITTHREAD pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32WAITTHREAD>,  
TP = .preDOS32WAITTHREAD,  
TYPE = (PRE,API),  
GROUP = SEM,  
DESC = "(OS) Dos32WaitThread Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,IF,4),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "ThreadID = <FMT_C_DWORD> WaitOption = <FMT_C_DWORD>"
```

*****DT postDOS32WAITTHREAD - DOS32WAITTHREAD post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32WAITTHREAD>,  
TP = .postDOS32WAITTHREAD,  
TYPE = (POST,API),  
GROUP = SEM,  
DESC = "(OS) Dos32WaitThread Post-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,IF,4),  
<LOG_C_RETVAL>,  
FMT = "ThreadID = <FMT_C_DWORD> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32IREAD - Dos32IRead pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32IREAD>,  
TP = .preDOS32IREAD,  
TYPE = (PRE,API),  
GROUP = FS,  
DESC = "(OS) Dos32IRead Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
```

```

MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM4>,D,4),
FMT = "File Handle = <FMT_C_DWORD> Buffer = <FMT_C_PTR>",
FMT = "Buffer Size = <FMT_C_DWORD> Pointer to Bytes Read = <FMT_C_PTR>"

```

***DT postDOS32IREAD - Dos32IRead post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32IREAD>,
TP = .postDOS32IREAD,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) Dos32IRead Post-Invocation",
<LOG_ECX>,
<LOG_C_RETVAL>,
FMT = "Bytes Read = <FMT_ECX> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32IWrite - Dos32IWrite pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32IWRITE>,
TP = .preDOS32IWRITE,
TYPE = (PRE,API),
GROUP = FS,
DESC = "(OS) Dos32IWrite Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM4>,D,4),
FMT = "File Handle = <FMT_C_DWORD> Buffer = <FMT_C_PTR>",
FMT = "Buffer Size = <FMT_C_DWORD> Pointer to Bytes Written = <FMT_C_PTR>"

```

***DT postDOS32IWRITE - Dos32IWrite post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32IWRITE>,
TP = .postDOS32IWRITE,
TYPE = (POST,API),
GROUP = FS,
DESC = "(OS) Dos32IWrite Post-Invocation",
<LOG_ECX>,
<LOG_C_RETVAL>,
FMT = "Bytes Written = <FMT_ECX> Return code = <FMT_C_RETVAL>"

```

***DT preDOS32DUMPPROCESS - Dos32DumpProcess pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOS32DUMPPROCESS>,
TP = .preDOS32DUMPPROCESS,
TYPE = (PRE,API),
GROUP = TK,
DESC = "(OS) Dos32DumpProcess Pre-Invocation",
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),
MEM = (<KSTK32_ARGS>+<C_PARM3>,D,4),
FMT = "Flag = <FMT_C_DWORD> Drive = <FMT_C_DWORD>",
FMT = "PID = <FMT_C_DWORD>"

```

***DT postDOS32DUMPPROCESS - Dos32DumpProcess post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOS32DUMPPROCESS>,
TP = .postDOS32DUMPPROCESS,
TYPE = (POST,API),

```

```
GROUP = TK,  
DESC = "(OS) Dos32DumpProcess Post-Invocation",  
<LOG_C_RETVAL>,  
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32SUPPRESSPOPUPS - Dos32SuppressPopUps pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOS32SUPPRESSPOPUPS>,  
TP = .preDOS32SUPPRESSPOPUPS,  
TYPE = (PRE,API),  
GROUP = TK,  
DESC = "(OS) Dos32SuppressPopUps Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM1>,D,4),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "Flag = <FMT_C_DWORD> Drive = <FMT_C_DWORD>"
```

*****DT postDOS32SUPPRESSPOPUPS - Dos32SuppressPopUps post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOS32SUPPRESSPOPUPS>,  
TP = .postDOS32SUPPRESSPOPUPS,  
TYPE = (POST,API),  
GROUP = TK,  
DESC = "(OS) Dos32SuppressPopUps Post-Invocation",  
<LOG_C_RETVAL>,  
FMT = "Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32QUERYEXTLIBPATH - Dos32QueryExtLIBPATH pre-invocation tp.**

```
TRACE MINOR = <RAS_PRE_DOS32QUERYEXTLIBPATH>,  
TP = .preDOS32QUERYEXTLIBPATH,  
TYPE = (PRE,API),  
GROUP = VM,  
DESC = "(OS) Dos32QueryExtLIBPATH Pre-Invocation",  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "Flags = <FMT_C_DWORD>"
```

*****DT postDOS32QUERYEXTLIBPATH - Dos32QueryExtLIBPATH post-invocation tp.**

```
TRACE MINOR = <RAS_POST_DOS32QUERYEXTLIBPATH>,  
TP = .postDOS32QUERYEXTLIBPATH,  
TYPE = (POST,API),  
GROUP = VM,  
DESC = "(OS) Dos32QueryExtLIBPATH Post-Invocation",  
ASCIIIZ= (<KSTK32_ARGS>+<C_PARM1>,IF,<MAXPATHLEN>),  
<LOG_C_RETVAL>,  
FMT = "Path = <FMT_C_ASCIIIZ> Return code = <FMT_C_RETVAL>"
```

*****DT preDOS32SETEXTLIBPATH - Dos32SetExtLIBPATH pre-invocation tp.**

```
TRACE MINOR = <RAS_PRE_DOS32SETEXTLIBPATH>,  
TP = .preDOS32SETEXTLIBPATH,  
TYPE = (PRE,API),  
GROUP = VM,  
DESC = "(OS) Dos32SetExtLIBPATH Pre-Invocation",  
ASCIIIZ= (<KSTK32_ARGS>+<C_PARM1>,IF,<MAXPATHLEN>),  
MEM = (<KSTK32_ARGS>+<C_PARM2>,D,4),  
FMT = "Path = <FMT_C_ASCIIIZ> Flags = <FMT_C_DWORD>"
```

*****DT postDOS32SETEXTLIBPATH - Dos32SetExtLIBPATH post-invocation tp.**

```
TRACE MINOR = <RAS_POST_DOS32SETEXTLIBPATH>,
TP      = .postDOS32SETEXTLIBPATH,
TYPE    = (POST,API),
GROUP   = VM,
DESC    = "(OS) Dos32SetExtLIBPATH Post-Invocation",
<LOG_C_RETVAL>,
FMT     = " Return code = <FMT_C_RETVAL>"
```

***DT preDOS32VERIFYPIDTID - Dos32VERIFYPIDTID pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOS32VERIFYPIDTID>,
TP      = .preDOS32VERIFYPIDTID,
TYPE    = (PRE,API),
GROUP   = TK,
DESC    = "(OS) Dos32VERIFYPIDTID Pre-Invocation",
MEM     = (<KSTK32_ARGS>+<C_PARM1>,D,4),
MEM     = (<KSTK32_ARGS>+<C_PARM2>,D,4),
FMT     = "Pid = <FMT_C_DWORD> Tid = <FMT_C_DWORD>"
```

***DT postDOS32VERIFYPIDTID - Dos32VERIFYPIDTID post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOS32VERIFYPIDTID>,
TP      = .postDOS32VERIFYPIDTID,
TYPE    = (POST,API),
GROUP   = TK,
DESC    = "(OS) Dos32VERIFYPIDTID Post-Invocation",
<LOG_C_RETVAL>,
FMT     = "Return code = <FMT_C_RETVAL>"
```

Task operations (IPC)

***DT preDOSMAKENMPIPE - DosMakeNmPipe pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSMAKENMPIPE>,
TP      = .preDOSMAKENMPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosMakeNmPipe Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+16,INDIRECT,<MAXPATHLEN>),
FMT     = "Name = <FMT_ASCIIIZ>"
```

***DT postDOSMAKENMPIPE - DosMakeNmPipe post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSMAKENMPIPE>,
TP      = .postDOSMAKENMPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosMakeNmPipe Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Handle = <FMT_AX> Return code = <FMT_RETVAL>"
```

***DT preDOSCALLNMPPIPE - DosCallNmPipe pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCALLNMPPIPE>,
TP      = .preDOSCALLNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosCallNmPipe Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+20,INDIRECT,<MAXPATHLEN>),
FMT     = "Name = <FMT_ASCIIIZ>"

```

***DT postDOSCALLNMPPIPE - DosCallNmPipe post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCALLNMPPIPE>,
TP      = .postDOSCALLNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosCallNmPipe Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Bytes out = <FMT_AX> Return code = <FMT_RETVAL>"

```

***DT preDOSCONNECTNMPPIPE - DosConnectNmPipe pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSCONNECTNMPPIPE>,
TP      = .preDOSCONNECTNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosConnectNmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"

```

***DT postDOSCONNECTNMPPIPE - DosConnectNmPipe post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSCONNECTNMPPIPE>,
TP      = .postDOSCONNECTNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosConnectNmPipe Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSDISCONNECTNMPPIPE - DosDisConnectMmPipe pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSDISCONNECTNMPPIPE>,
TP      = .preDOSDISCONNECTNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosDisConnectMmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"

```

***DT postDOSDISCONNECTNMPPIPE - DosDisConnectMmPipe post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSDISCONNECTNMPPIPE>,
TP      = .postDOSDISCONNECTNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosDisConnectMmPipe Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSPEEKNMPPIPE - DosPeekNmPipe pre-invocation trace point

```
TRACE MINOR = <RAS_PRE_DOSPEEKNMPIPE>,
TP      = .preDOSPEEKNMPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosPeekNmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSPEEKNMPIPE - DosPeekNmPipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSPEEKNMPIPE>,
TP      = .postDOSPEEKNMPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosPeekNmPipe Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Bytes read = <FMT_AX> Return code = <FMT_RETVAL>"
```

*****DT preDOSQNMPHANDSTATE - DosQNmPHandState pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQNMPHANDSTATE>,
TP      = .preDOSQNMPHANDSTATE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPHandState Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSQNMPHANDSTATE - DosQNmPHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQNMPHANDSTATE>,
TP      = .postDOSQNMPHANDSTATE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPHandState Post-Invocation",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Handle state = <FMT_AX> Return code = <FMT_RETVAL>"
```

*****DT preDOSQNMPIPEINFO - DosQNmPipeInfo pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSQNMPIPEINFO>,
TP      = .preDOSQNMPIPEINFO,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPipeInfo Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSQNMPIPEINFO - DosQNmPipeInfo post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSQNMPIPEINFO>,
TP      = .postDOSQNMPIPEINFO,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPipeInfo Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSRAWREADNMPIPE - DosRawReadNmPipe pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSRAWREADNMPPIPE>,
TP      = .preDOSRAWREADNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosRawReadNmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSRAWREADNMPPIPE - DosRawReadNmPipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSRAWREADNMPPIPE>,
TP      = .postDOSRAWREADNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosRawReadNmPipe Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Bytes read = <FMT_CX> Return code = <FMT_RETVAL>"
```

*****DT preDOSRAWWRITENMPPIPE - DosRawWriteNmPipe pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSRAWWRITENMPPIPE>,
TP      = .preDOSRAWWRITENMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosRawWriteNmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSRAWWRITENMPPIPE - DosRawWriteNmPipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSRAWWRITENMPPIPE>,
TP      = .postDOSRAWWRITENMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosRawWriteNmPipe Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Bytes written = <FMT_CX> Return code = <FMT_RETVAL>"
```

*****DT preDOSSETNMPHANDSTATE - DosSetNmPHandState pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETNMPHANDSTATE>,
TP      = .preDOSSETNMPHANDSTATE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosSetNmPHandState Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSSETNMPHANDSTATE - DosSetNmPHandState post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETNMPHANDSTATE>,
TP      = .postDOSSETNMPHANDSTATE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosSetNmPHandState Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSTRANSACTNMPPIPE - DosTransactNmPipe pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSTRANSACTNMPPIPE>,
TP      = .preDOSTRANSACTNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosTransactNmPipe Pre-Invocation",
<LOG_BX>,
FMT     = "Handle = <FMT_BX>"
```

*****DT postDOSTRANSACTNMPPIPE - DosTransactNmPipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSTRANSACTNMPPIPE>,
TP      = .postDOSTRANSACTNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosTransactNmPipe Post-Invocation",
<LOG_CX>,
<LOG_RETVAL>,
FMT     = "Bytes out = <FMT_CX>  Return code = <FMT_RETVAL>"
```

*****DT preDOSWAITNMPPIPE - DosWaitNmPipe pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSWAITNMPPIPE>,
TP      = .preDOSWAITNMPPIPE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosWaitNmPipe Pre-Invocation",
ASCIIIZ=(<KSTK_ARGS>+<SKIP_DWORD>,INDIRECT,<MAXPATHLEN>),
FMT     = "Name = <FMT_ASCIIIZ>"
```

*****DT postDOSWAITNMPPIPE - DosWaitNmPipe post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSWAITNMPPIPE>,
TP      = .postDOSWAITNMPPIPE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosWaitNmPipe Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSSETNMPPIPESEM - DosSetNmPipeSem pre-invocation trace point**

```
TRACE MINOR = <RAS_PRE_DOSSETNMPPIPESEM>,
TP      = .preDOSSETNMPPIPESEM,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosSetNmPipeSem Pre-Invocation",
<LOG_AX>,
<LOG_CX>,
<LOG_BX>,
FMT     = "Semaphore handle = <FMT_AX><FMT_CX>  Pipe handle = <FMT_BX>"
```

*****DT postDOSSETNMPPIPESEM - DosSetNmPipeSem post-invocation trace point**

```
TRACE MINOR = <RAS_POST_DOSSETNMPPIPESEM>,
TP      = .postDOSSETNMPPIPESEM,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosSetNmPipeSem Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"
```

*****DT preDOSQNMPIPESEMSTATE - DosQNmPipeState pre-invocation trace point**

```

TRACE MINOR = <RAS_PRE_DOSQNMPIPESEMSTATE>,
TP      = .preDOSQNMPIPESEMSTATE,
TYPE    = (PRE,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPipeState Pre-Invocation",
<LOG_AX>,
<LOG_BX>,
FMT     = "Semaphore handle = <FMT_AX><FMT_BX>"

```

***DT postDOSQNMPIPESEMSTATE - DosQNmPipeState post-invocation trace point

```

TRACE MINOR = <RAS_POST_DOSQNMPIPESEMSTATE>,
TP      = .postDOSQNMPIPESEMSTATE,
TYPE    = (POST,API),
GROUP   = PIP,
DESC    = "(OS) DosQNmPipeState Post-Invocation",
<LOG_RETVAL>,
FMT     = "Return code = <FMT_RETVAL>"

```

***DT preDOSMUXSEMWAIT - DosMuxSemWait pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSMUXSEMWAIT>,
TP      = .DOSMUXSEMWAIT,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosMuxSemWait Pre-Invocation",
MEM     = (RSS+SP+8,D,4), /* ArgVar TimeOut - skip CS:EIP. */
FMT     = "Timeout=<FMT_C_DWORD>"

```

***DT postDOSMUXSEMWAIT2 - DosMuxSemWait post-invocation trace point (2)

```

TRACE MINOR = <RAS_POST_DOSMUXSEMWAIT2>,
TP      = .postDOSMUXSEMWAIT2,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosMuxSemWait Post-Invocation (No Wait)",
MEM     = (RSS+SP+8+<SKIP_DWORD>+<SKIP_PTR>, I, 2),
REGS    = (AX),
FMT     = "Index = <FMT_C_WORD> Return Code = <FMT_WORD>"

```

***DT postDOSMUXSEMWAIT - DosMuxSemWait post-invocation trace point (1)

```

TRACE MINOR = <RAS_POST_DOSMUXSEMWAIT>,
TP      = .postDOSMUXSEMWAIT,
TYPE    = (PRE,API),
GROUP   = SEM,
DESC    = "(OS) DosMuxSemWait Post-Invocation (Waited)",
<LOG_AX>,
<LOG_RETVAL>,
FMT     = "Index = <FMT_AX> Return Code = <FMT_RETVAL>"

```

***DT preDOSHOLDSIGNAL - DosHoldSignal pre-invocation trace point

```

TRACE MINOR = <RAS_PRE_DOSHOLDSIGNAL>,
TP      = .preDOSHOLDSIGNAL,
TYPE    = (PRE,API),
GROUP   = SIG,
DESC    = "(OS) DosHoldSignal Pre-Invocation",
REGS    = (AX),
FMT     = "Action = <FMT_WORD>"

```

***DT post2DOSHOLDSIGNAL - DosHoldSignal post-invocation trace point 2

```
TRACE MINOR = <RAS_POST2_DOSHOLDSIGNAL>,
TP          = .post2DOSHOLDSIGNAL,
TYPE       = (POST,API),
GROUP      = SIG,
DESC       = "(OS) DosHoldSignal Post-Invocation (2)",
REGS       = (AX),
FMT        = "Return code = <FMT_WORD>"
```

***DT postDOSHOLDSIGNAL - DosHoldSignal post-invocation trace point

```
TRACE MINOR = <RAS_POST_DOSHOLDSIGNAL>,
TP          = .postDOSHOLDSIGNAL,
TYPE       = (POST,API),
GROUP      = SIG,
DESC       = "(OS) DosHoldSignal Post-Invocation",
           <LOG_RETVAL>,
FMT        = "Return code = <FMT_RETVAL>"
```

***DT postSchedNext - SchedNext post-invocation internal trace point

```
TRACE MINOR = <RAS_POST_INT_SCHEDNEXT>,
TP          = .postSchedNext,
TYPE       = (POST,INT),
GROUP      = TK,
DESC       = "(OS) Thread Reschedule Post-Invocation",
REGS       = (AX,DX),
FMT        = "Switched to PID = <FMT_WORD>  TID = <FMT_WORD>"
```

Notices

Third Edition (November 2003)

This document contains IBM proprietary information provided to you solely for the purpose of developing your own OS/2 software. In no event shall IBM be liable for any damage arising from use or misuse of this information. This document is distributed without any warranties, including, but not limited to, the warranty of merchantability and fitness for a particular purpose.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

(C) Copyright 1987-1992, International Business Machines Corporation.

(C) Copyright 1987-1991, Microsoft Corporation.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
